UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

Tesis para optar al Título de
MAGISTER EN INFORMÁTICA
PEDECIBA

VECTOR REPRESENTATION OF INTERNET DOMAIN NAMES USING WORD EMBEDDING TECHNIQUES

Author: Waldemar Joel López Anzolabehere
Advisor: Dr. Pablo Rodríguez-Bocca

Montevideo, Uruguay
2019

PÁGINA DE APROBACIÓN

UNIVERSIDAD DE LA REPÚBLICA
FACULTAD DE INGENIERÍA

El tribunal docente integrado por los abajo firmantes aprueba la Tesis de Investigación**:** *"Vector representation of Internet domain names using word embedding techniques".*

Autor: Ing. Waldemar Joel López Anzolabehere
Tutor: Dr. Pablo Rodríguez-Bocca
Carrera: Maestría en Informática (PEDECIBA)

Puntaje: .....

Tribunal

Profesor ..... (Nombre y firma)

Profesor ..... (Nombre y firma)

Profesor ..... (Nombre y firma)

Fecha:

AGRADECIMIENTOS

Quiero agradecer al *PEDECIBA* y a la Facultad de Ingeniería de la *UDELAR* por permitirme realizar este trabajo, apoyándome para lograr los resultados y poder publicar los avances realizados.

Un especial agradecimiento a mi tutor Pablo Rodríguez quién en todo momento supo guiarme y marcar el norte, principalmente en aquellos momentos donde el camino se hacía cuesta arriba. Su disponibilidad y buena predisposición fueron fundamentales para que este trabajo llegara a buen puerto.

También agradecer a Jorge Merlino por los intercambios de ideas y discusiones en torno a este trabajo. Espero que los humildes resultados logrados en esta tesis puedan ser tomados y extendidos como parte de su doctorado.

Quiero dar las gracias a los autores de *App2Vec*, en particular a Qiann Ma por ceder muy gentilmente el código con la adaptación que realizaron al método *CBOW* de *Word2Vec*, el cual fue parte de los métodos de word embedding evaluados en este trabajo.

Agradezco también a los doctores Laura Alonso i Alemany, Aiala Rosá Furman y Claudio Risso que me brindan el honor de juzgar este trabajo.

No puedo dejar de agradecer a la educación pública de mi país por haberme formado desde la primaria. A todas mis maestras, profesores, profesoras y docentes que con un granito de arena contribuyeron a mi curiosidad y amor por las ciencias. A ellos un enorme gracias.

Mi mayor reconocimiento a mis padres, quienes me educaron y transmitieron los valores de los cuales me siento orgullo. Ellos me enseñaron que no hay tesoro mas preciado que el conocimiento y la educación. Predicando con el ejemplo me mostraron que con honestidad, trabajo, esfuerzo y constancia es posible hacer frente a cualquier adversidad y lograr cualquier objetivo.

Por último, agradezco infinitamente el amor incondicional de mi esposa e hijos, por su apoyo continuo, por su generosidad comprendiéndome y regalándome mucho de su tiempo, por haber sido mi compañía imprescindible para terminar este viaje.

Resumen

La vectorización de palabras es un conjunto de técnicas bien conocidas y ampliamente usadas en el procesamiento del lenguaje natural (*PLN*). Esta tesis explora el uso de vectorización de palabras en un nuevo escenario. Un modelo de espacio vectorial (*VSM*) para nombres de dominios de Internet (*DNS*) es creado tomando ideas fundamentales de *PLN,* las cuales son aplicadas a consultas reales anonimizadas de logs de *DNS* de un gran proveedor de servicios de Internet (*ISP*). El objetivo principal es encontrar dominios relacionados semánticamente solamente usando información de consultas *DNS* sin ningún otro conocimiento sobre el contenido de esos dominios.

Un conjunto de transformaciones a través de un detallado pipeline de preprocesamiento con ocho pasos específicos es definido para llevar el problema original a un problema en el campo de *PLN*. Una vez aplicado el pipeline de preprocesamiento y los logs de *DNS* son transformados a un corpus de texto estándar, se muestra que es posible utilizar con éxito técnicas del estado del arte respecto a vectorización de palabras para construir lo que denominamos un *DNS-VSM* (un modelo de espacio vectorial para nombres de dominio de Internet).

Diferentes técnicas de vectorización de palabras son evaluadas en este trabajo: *Word2Vec* (con arquitectura *Skip-Gram* y *CBOW*), *App2Vec* (con arquitectura *CBOW* y agregando intervalos de tiempo entre consultas *DNS*), y *FastText* (incluyendo información a nivel de sub-palabra).
Los resultados obtenidos se comparan usando varias métricas de la teoría de Recuperación de Información y la calidad de los vectores aprendidos es validada por una fuente externa, un servicio para obtener sitios similares ofrecido por *Alexa Internet, Inc*[1].
Debido a características intrínsecas de los nombres de dominio, encontramos que *FastText* es la mejor opción para construir un modelo de espacio vectorial para *DNS*. Además, su performance es comparada contra dos métodos de línea base: *Random Guessing* (devolviendo cualquier nombre de dominio del dataset de forma aleatoria) y *Zero Rule*

[1] https://www.alexa.com/

(devolviendo siempre los mismos dominios más populares), superando a ambos de manera considerable.

Los resultados presentados en este trabajo pueden ser útiles en muchas actividades de ingeniería, con aplicación práctica en muchas áreas. Algunos ejemplos incluyen recomendaciones de sitios web, análisis competitivo, identificación de sitios riesgosos o fraudulentos, sistemas de control parental, mejoras de *UX* (basada en recomendaciones, corrección ortográfica, etc.), análisis de flujo de clics, representación y clustering de perfiles de navegación de usuarios, optimización de sistemas de cache en resolutores de *DNS* recursivos (entre otros).

Por último, como contribución a la comunidad académica, un conjunto de vectores del *DNS-VSM* entrenado sobre un juego de datos similar al utilizado en esta tesis es liberado y hecho disponible para descarga a través de la página github en [1]. Con esto esperamos a que más trabajos e investigaciones puedan realizarse usando estos vectores.

Palabras clave: DNS, *VSM, Vectorización de palabras, Word2vec, FastText, App2vec, Similitud semántica, Procesamiento de Lenguaje Natural (PLN).*

Abstract

Word embeddings is a well-known set of techniques widely used in natural language processing (*NLP*). This thesis explores the use of word embeddings in a new scenario. A vector space model (*VSM*) for Internet domain names (*DNS)* is created by taking core ideas from *NLP* techniques and applying them to real anonymized *DNS* log queries from a large Internet Service Provider (*ISP*). The main goal is to find semantically similar domains only using information of *DNS* queries without any other knowledge about the content of those domains.

A set of transformations through a detailed preprocessing pipeline with eight specific steps is defined to move the original problem to a problem in the *NLP* field. Once the preprocessing pipeline is applied and the *DNS* log files are transformed to a standard text corpus, we show that state-of-the-art techniques for word embeddings can be successfully applied in order to build what we called a *DNS-VSM* (a vector space model for Internet domain names).

Different word embeddings techniques are evaluated in this work: *Word2Vec* (with *Skip-Gram and CBOW* architectures), *App2Vec* (with a *CBOW* architecture and adding time gaps between *DNS* queries), and *FastText* (which includes sub-word information).
The obtained results are compared using various metrics from Information Retrieval theory and the quality of the learned vectors is validated with a third party source, namely, similar sites service offered by *Alexa Internet, Inc*[2].
Due to intrinsic characteristics of domain names, we found that *FastText* is the best option for building a vector space model for *DNS*. Furthermore, its performance (considering the top 3 most similar learned vectors to each domain) is compared against two baseline methods: *Random Guessing* (returning randomly any domain name from the dataset) and *Zero Rule* (returning always the same most popular domains), outperforming both of them considerably.

[2] https://www.alexa.com/

The results presented in this work can be useful in many engineering activities, with practical application in many areas. Some examples include websites recommendations based on similar sites, competitive analysis, identification of fraudulent or risky sites, parental-control systems, *UX* improvements (based on recommendations, spell correction, etc.), click-stream analysis, representation and clustering of users navigation profiles, optimization of cache systems in recursive *DNS* resolvers (among others).

Finally, as a contribution to the research community a set of vectors of the *DNS-VSM* trained on a similar dataset to the one used in this thesis is released and made available for download through the github page in [1]. With this we hope that further work and research can be done using these vectors.

Keywords: DNS, *VSM, Word embeddings, Word2vec, FastText, App2vec, Semantic Similarity, Natural Language Processing (NLP).*

Contents

1. Chapter I - Introduction

In this first chapter, an introduction with the motivation, goals and contributions of this thesis is presented. Finally, the organization of the document is outlined.

1.1. Motivation and goals

The amount of time that people spend online has systematically increased in recent years [2]. To understand the behavior of users in online content consumption is the focus of several research. It has large implications to network design, online business, and media industry [3].

Many studies apply machine learning to historical patterns of network resource consumption in order to extract knowledge about online customer behavior [4], [5]. Due to the inaccessibility of the information, few of these studies use the traces of *DNS* queries for this purpose. The few exceptions are [6], [7], [8], [9], [10], where none of them has as main objective to extract knowledge about the semantic nature of the queried domains.

There are several Web tools that try to estimate the semantic similarity between sites. For example to provide web site owners the possibility to find competitors for the same target audience, and to advice end-users on alternative providers for the same content. As we will see in Section 2.2.5, these solutions and strategies have lot of disadvantages.

In this work, a novel approach to address the problem of finding similarities between Internet domain names (without suffering from the previous mentioned disadvantages) is presented. The main goals are:

I. Define a *similarity measure* between domain names only using information of *DNS* queries (without any other previous knowledge about the content of those domains).

II. Given any domain name and using the defined similarity function, find other semantically and syntactically related domain names.
III. Find a way to identify other complex relationships between domain names, for example: complementary domain names, domain names that are generally accessed together or relative relationships like *domain A* is to *domain B* in the same way that *domain C* is to *domain D*.

Many use cases can benefit from a solution that satisfy these goals. Besides the common uses cases that we can see nowadays that include websites recommendations based on similar sites or competitive analysis, many others applications such as identification of fraudulent or risky sites, parental-control systems, *UX* improvements (based on recommendations, spell correction, etc), click-stream analysis, representation and clustering of users navigation profiles, optimization of cache systems in recursive *DNS* resolvers, and more, could leverage the results of this work.

1.2. Contributions

We can summarize the main contributions of this thesis in the following points:

- **State of the Art**
  A complete review of the most common techniques that are used for building *vector representation of words* is presented. Starting from the most basics ones that use sparse representations, to the most complex ones that are able to learn dense vectors representation using neural networks as part of a statistical language modelling. This review shows the natural evolution of these techniques and it is by itself a good survey of the bibliography about the topic.

- **Mapping and transformation of DNS data to a problem in the NLP field**
  A mapping between concepts that come from the *NLP* field to concepts in the *DNS* system is identified. Additionally, a set of

characteristics and limitations about the *DNS* data and how the *DNS* system works are highlighted and a set of transformations through a detailed preprocessing pipeline with eight specific steps is defined to move the original problem to a problem in the *NLP* field.

- **Vector Space Model for Domain Names (DNS-VSM)**
  This thesis shows that once the preprocessing pipeline is applied and the *DNS* log files are transformed to a standard text corpus in the *NLP* field then, *state-of-the art* techniques for *word embeddings* can be successfully applied to the corpus in order to build what we called a *DNS-VSM* (a vector space model for domain names).
  In our *DNS-VSM* domain names are represented by vectors where related domain names are mapped to nearby points in the high dimensional space. This *DNS-VSM* is built only using information of *DNS* queries without any other previous knowledge about the content hosted in each domain.

- **Pre-trained vectors for the DNS-VSM**
  A set of vectors of the *DNS-VSM* (trained on a similar dataset to the one used in this thesis) is released and made available for download through the github page in [1]. With this, we hope that further work can be done using these vectors.

1.3. Publications and conferences

The following publications were generated during this work:

- W. Lopez, J. Merlino and P. Rodriguez-Bocca, "Vector representation of internet domain names using a word embedding technique," 2017 XLIII Latin American Computer Conference (CLEI), Cordoba, 2017, pp. 1-8.

- W. Lopez, J. Merlino and P. Rodriguez-Bocca, "Extracting semantic information from Internet Domain Names using word embeddings", submitted to Engineering Applications of Artificial Intelligence (ELSEVIER), 2019.

1.4. Summary and organization of the document

This thesis is structured in three parts, subdivided in chapters. The remainder of this document is organized as follows:

Part I - STATE OF THE ART
In Chapter 2 the required background about how *DNS* systems work is presented (Section 2.1) as well as the related work regarding current solutions for finding semantic similarity for Internet domain names (Section 2.2). In the end of this Chapter a set of disadvantages of the current solutions are highlighted, therefore motivating the exploration of the other approaches considered in this work.
Then, Chapter 3 introduces the most common techniques that are used for building vector representation of words and documents, starting from the most basics ones that use sparse representations, to the most complex ones that are able to learn dense vectors representation using neural networks as part of a statistical language modelling. In particular, Sections 3.2.2.3, 3.2.2.4 and 3.2.2.5 present all the theory behind *Word2Vec*, *App2Vec* and *FastText* respectively, which are used later as a core block for the novel approach presented in this work to address the problem of finding semantically related domain names.

Part II - DNS VECTOR SPACE MODEL
In Chapter 4 all the details for building a vector space model for Internet domain names (*DNS-VSM*) are presented. Firstly, in Section 4.1 a descriptive analysis of the data used to build the different models is shown. Then, in Section 4.2 the preprocessing steps for building the dataset is described, and in Section 4.3 the evaluation framework to be used is presented as well as the baseline models. In Section 4.4 the first model based on *Word2Vec* is described in details. In Section 4.5 the addition of the time factor (the elapsed time gap between two consecutives *DNS* queries requested by a same user) is studied and a second model based on *App2Vec* is evaluated. A final improvement by considering sub-word information is described in Section 4.6 where the last model based on *FastText* is presented. Finally, a summary analyzing the results and discussing possible use cases for the *DNS-VSM* is presented in Section 4.7.

Part III - CONCLUSIONS

Chapter 5 summarizes the main conclusions of this work and gives some ideas about possible directions of future work with the *DNS-VSM*.

Part I
STATE OF THE ART

2. Chapter Ⅱ - Domain Names on Internet

2.1. Basic concepts about DNS

The *Domain Name System (DNS)* [11], [12] is a decentralized service for naming computers and other resources in a network. Each of these resources is assigned a domain name which is a hierarchical string defining a node in a tree structure. The domain name is formed by the labels of the nodes in the tree traversed from the leaf node to the root node, separated by points, as it is shown in the following example:

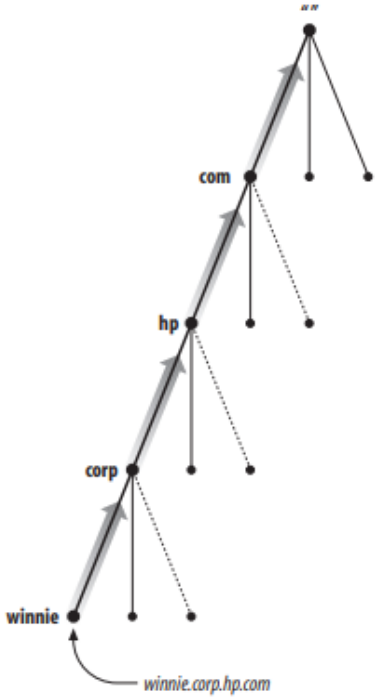


Figure 1 - Reading the fully qualified domain name (FQDN) winnie.corp.hp.com from the leaf node to the root node. Source [13].

As we can see in *Figure 1*, the *FQDN winnie.corp.hp.com* can be splitted into different nodes/labels, with the following characteristics:

- A null label, or " " is reserved for the root node. In text, the root node is written as a single dot (.) [13].

- The "*com"* label represents the first/top-level domain *(TLD)*. Other commonly used TLDs are the generic TLDs *(gTLDs)* like *edu, org. mil, int, net* or the country code TLDs *(ccTLDs)* like *us, de, br, uy, etc.* (a list of all valid top-level domains[3] is maintained by the IANA[4] and is updated from time to time*)*.

- The "*hp"* label represents the *second-level domain* and commonly refers to the organization (in this example it would be *Hewlett-Packard*) that registered the domain name with a *domain name registrar.*

- The *"corp"* label is a subdomain of "*hp.com"* and it could represent any relevant concept for the domain *("corp" could be used by the organization to identify its corporate headquarters, for example).* One domain can contain several subdomains, for example, *www.example.com*, *ftp.example.com* and *smtp.example.com* all are subdomains of *example.com* (a web server, a file transfer server and mail server respectively)

- *"winnie"* label is just a simple *hostname*, and it is a subdomain of *corp.hp.com*

The *DNS* system has been in use in the Internet since 1985 and is one of the most essential services in the network. It is decentralized as the responsibility for resolving each component of the domain name is delegated to a different name server, thus avoiding a single central database and a single point of failure. Also, there could be several domain servers to resolve the same domain, providing thus a fault tolerant configuration.

[3] http://data.iana.org/TLD/tlds-alpha-by-domain.txt

[4] https://www.iana.org/

In the Internet, the most fundamental service provided by *DNS* is to translate easily memorized domain names (human-readable) to IP addresses. Each domain can contain different subdomains with different types. The most common types are shown in *Table 1*.

Type	***Description***	***Example***
A	IPv4 address translation	Host1.example.mydomain.com. IN A 127.0.0.1
AAAA	IPv6 address translation	ipv6_host1.example.mydomain.com. IN AAAA 4321:0:1:2:3:4:567:89ab
MX	SMTP mail exchangers	example.mydomain.com. MX 10 mailserver1.example.mydomain.com
NS	Other name servers	example.mydomain.com. IN NS nameserver1.example.mydomain.com
CNAME	Domain name aliases	aliasname.example.mydomain.com. CNAME truename.example.mydomain.com.
PTR	Reverse DNS queries, for example to query the domain name for a given IP address	1.0.0.10.in-addr.arpa. PTR host.example.mydomain.com.

Table 1 - Most common DNS record types

Each domain has at least one *authoritative name server* that contains the original information about the domain and its subdomains. An authoritative name server only gives answers to DNS *queries* from data that has been configured on that server. Potentially, an authoritative name server could delegate a subdomain to other authoritative servers building a hierarchical tree of authorities.

On the top of the hierarchy are the *root DNS servers*. There are 13 *logical root name servers*, which form a network of hundreds of servers in many countries around the world. *Table 2* shows the root servers and its operators.

HOSTNAME	***IP ADDRESSES***	***MANAGER***
a.root-servers.net	198.41.0.4, 2001:503:ba3e::2:30	VeriSign, Inc.
b.root-servers.net	199.9.14.201, 2001:500:200::b	University of Southern California (ISI)
c.root-servers.net	192.33.4.12, 2001:500:2::c	Cogent Communications
d.root-servers.net	199.7.91.13, 2001:500:2d::d	University of Maryland
e.root-servers.net	192.203.230.10, 2001:500:a8::e	NASA (Ames Research Center)
f.root-servers.net	192.5.5.241, 2001:500:2f::f	Internet Systems Consortium, Inc.
g.root-servers.net	192.112.36.4, 2001:500:12::d0d	US Department of Defense (NIC)
h.root-servers.net	198.97.190.53, 2001:500:1::53	US Army (Research Lab)
i.root-servers.net	192.36.148.17, 2001:7fe::53	Netnod
j.root-servers.net	192.58.128.30, 2001:503:c27::2:30	VeriSign, Inc.
k.root-servers.net	193.0.14.129, 2001:7fd::1	RIPE NCC
l.root-servers.net	199.7.83.42, 2001:500:9f::42	ICANN
m.root-servers.net	202.12.27.33, 2001:dc3::35	WIDE Project

Table 2 - List of Root Servers[5]

On the other hand, there are *recursive DNS servers* (generally provided by the *ISP*) which are capable to resolve queries about domain names by means of recursive queries to possibly several authoritative name servers starting from the root servers.
In this way, root *DNS* servers delegates authoritative subdomains to *ccTLDs* and *gTLDs* servers which are responsible to populate country domain names and generic domain names respectively. Similarly, these top-level domain servers generally delegate subdomains to other

[5] https://www.iana.org/domains/root/servers

organizations, which are free to continue delegating. All this chain of delegations builds a tree of authoritative name servers, where each record has a specific location. Hence, in order to resolve a name, the tree must be traversed, from the root to the most specific authoritative server, which should have the answer (if it exists).

The client components of the *DNS* system are called *DNS resolvers*. Resolvers usually query recursive servers to find an answer, which generally need to perform many iterative queries to other name servers until find the answer. *Figure 2* illustrates this resolution process.
Once the answer for a query is found, the recursive server typically saves the response in a local cache in order to increase efficiency, avoiding to repeat all the same process for domain names that were previously queried. The duration of the cached data depends on the *TTL (time to live)* configuration of each domain at the authoritative servers.

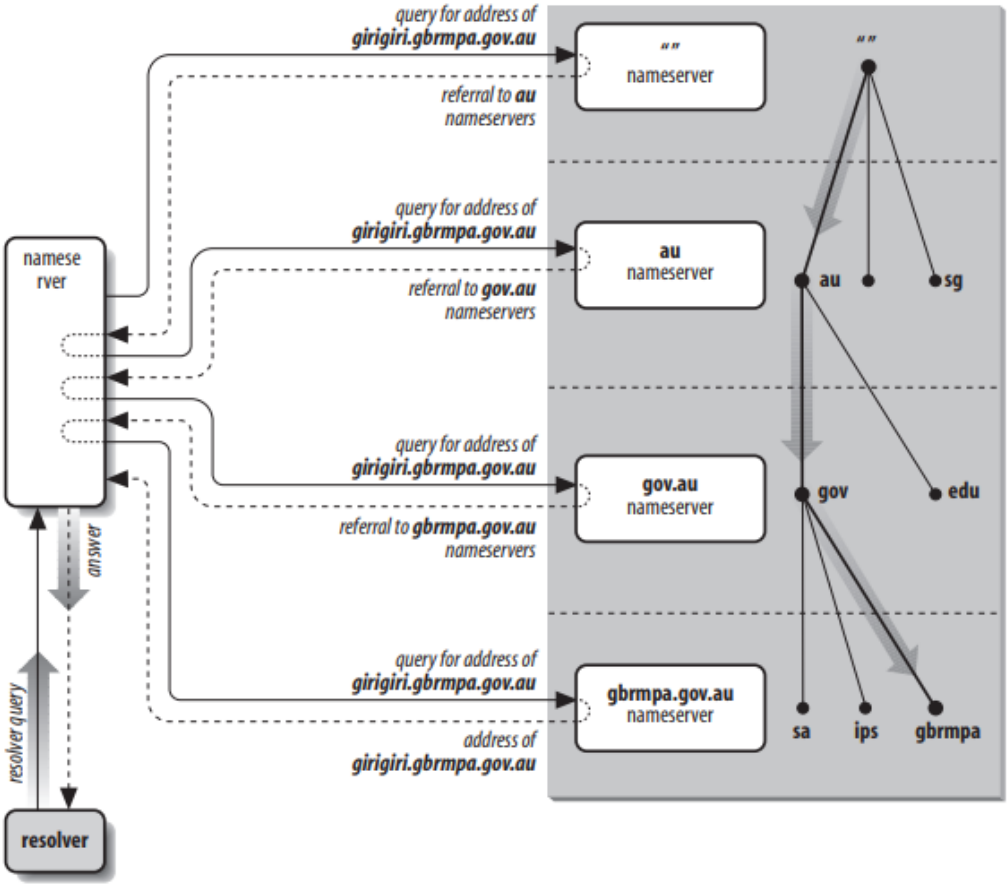


Figure 2 - Resolution of girigiri.gbrmpa.gov.au on the Internet. Source [13].

In summary, the *DNS* system is a critical service on the Internet, essential for most applications (Web, Mail, etc.) to work. The system is a distributed database of records, being the most important those that transform names into IP addresses. The database is structured in an arborescent manner in authoritative servers, which are administered autonomously by various institutions "owners" of a possible portion of subdomains.
Clients, who perform record queries, use a special application, called resolver, included in operating systems. However, the resolvers do not consult the database directly, but send the query to recursive servers who traverse the tree, keep the result for future potential queries, and respond to the client. The recursive servers then have a trace of queries for each client, and it is the objective of this work to extract knowledge of these traces.
In Chapter 4 we will see how to transform these traces and use them with *NLP* algorithms to extract semantic information about domain names. But before that, let's see what are the current approaches to find semantic similarity between domain names and what are the main issues they present.

2.2. Semantic similarity for Internet Domain Names

So far, we have defined what *DNS* is and how domain names are resolved by *DNS* servers. We have seen, that *DNS* servers can solve a *DNS* query iteratively or recursively, but in none of those cases *DNS* servers know really what kind of business need or goal a domain name has. Domain names are seen by *DNS* servers just as strings composed of labels with a hierarchical structure, and no relevant or semantic information is known a priori from those strings.

Other thing that we have noticed in the previous section is that *DNS* servers typically cache the results of *DNS* queries in order to optimize the overall performance of the *DNS* server. Probably, if semantic information about *DNS* was known in advance, more powerful cache strategies could be possible to implement, for example by loading similar domains or complementary domains that are commonly consumed together.

Knowing the semantic of domain names could also allow to implement more secure systems, warning about possible fraudulent sites for example, or detecting sites that could contain risky contents. Clustering of domain names according to the kind of content that they provide, would allow to recommend similar contents, identify competitors, block inadequate contents in parental control agents (among others). Hence, we realize that enriching domain names with semantic information would bring lot of possibilities.

The term semantic similarity is usually employed over sets of words (texts) to measure the likeness of their meaning. In the context of this work we will define the semantic similarity between domain names, as the distance between their semantic content. For example, two news providers are expected to be semantically similar, as well as two retail stores, and if they offer the same category of products should be closer.

In the rest of this section an overview of current mechanisms and tools to identify semantic similarity between domain names is presented. Later, the necessity of new methods is pointed out by understanding the disadvantages of the current strategies, and thus, motivating the main work of this thesis that proposes a novel methodology to obtain the semantic information associated to domain names using only available information in the *DNS* recursive servers.

2.2.1. Alexa

Alexa Internet, Inc.[6], founded in 1996 as an independent company and acquired later by *Amazon* in 1999, has been one of the most important references in what is related to web analytics. Its *Alexa Rank* (a famous and widely used metric to measure the popularity of a website) it has received historically (and even today), a considerable importance by sites' owners and people working on search engine optimization (*SEO*).

In order to get all the data needed for its analytics services, *Alexa* collects information directly from multiple web browser extensions, toolbars

[6] https://www.alexa.com/

and also from sites that install a script that sends information to *Alexa's* servers (similar to *Google Analytics* [7]).
Toolbars and extensions are available for *Firefox*[8] and *Chrome*[9] and they are designed in such way that users are motivated to install and use them in order to access to features such as:

- *Alexa Traffic Rank* (see how popular a website is)
- *Related Links* (find sites that are similar to the site you are visiting)
- *Wayback* (see how a site looked in the past)
- *Search Analytics* (find out which queries drive traffic to a site)

With all this information collected and centralized, the analytics tools provided by *Alexa*, allow enterprise customers to find meaningful information about their sites.
These tools are divided in two main groups: *SEO tools*[10] (focused on keyword research and website optimization for Improving search engine rankings and results) and *Competitive Analysis Tools*[11] (focused on website analysis and market research for understanding the competitive landscape, track the performance of other sites and get visibility into competitor strategies).

In particular, one of the *Competitive Analysis Tools* is the *Audience Overlap Tool*[12]. Among other things, it allows to see clusters of related sites as it is shown in *Figure 3*. This is very interesting and related to this work, since in some way or another, semantic information start to appear when analyzing these clusters.
And going deeper, this tool offers a self contained service called *Find Similar Sites* [13] which allows to query for a specific domain name in order to know possible competitors based on similar sites. In its paid version, this service retrieves the top 100 most similar sites for a specific input domain name.

[7] https://analytics.google.com/
[8] https://www.alexa.com/toolbar?browser=firefox
[9] https://www.alexa.com/toolbar?browser=chrome
[10] https://try.alexa.com/marketing-stack/seo-tools/
[11] https://try.alexa.com/marketing-stack/competitive-analysis-tools
[12] https://try.alexa.com/marketing-stack/audience-overlap-tool
[13] https://www.alexa.com/find-similar-sites

Figure 4 shows a web view of the service displaying the top 5 most similar sites for *amazon.com*. The result list has a default order by the *Overlap Score* value. The meaning of this value according to *Alexa* is: *"the relative level of visitor (audience) overlap between any site and the target site. A site with a higher score shows higher audience overlap than a site with a lower score"*. This order will be important when defining the evaluation strategy explained in Section 4.3.

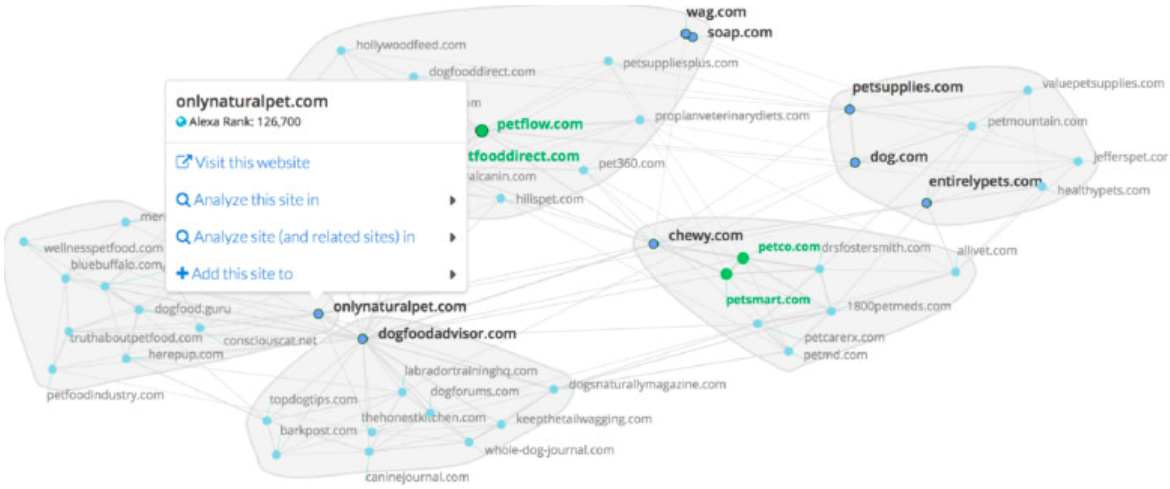


Figure 3 - Clusters of related sites (image credits: Alexa[14]).

Site	Overlap Score	Alexa Rank
amazon.com	-	**10**
ebay.com	71	41
reddit.com	41	18
walmart.com	38	178
wikipedia.org	37	5
pinterest.com	35	76

Figure 4 - Similar Sites for amazon.com according to Alexa. Retrieval date: Oct, 2018.

[14] https://try.alexa.com/offer/guided-tour/find-and-reach-your-audience

As explained before, the main objective of this thesis is to build a semantic similarity representation between domain names only using information of *DNS* queries from recursive *DNS* servers, without any other previous knowledge about the content of those domains.
In order to do this, some external sources will be used as reference for comparison. In particular (as it will be explained in Section 4.3, when describing the evaluation framework to use), the API service for finding similar sites provided by *Alexa* will be used for measuring the accuracy of the solution.

2.2.2. SimilarWeb

SimilarWeb[15], is an online competitive intelligence tool owned by the Israeli start-up *SimilarGroup*. The online tool as well as its browser extension[16] (available for *Chrome*, *Firefox*, *Safari* and *Opera*) shows many statistics about websites, including traffic sources, organic versus paid search, social traffic, related sites (the most related to this work), and more.

As it is shown in *Figure 5*, the main features are divided into nine different categories (see [14] or [15] for a detailed review):

- *General Overview* (in-depth traffic and engagement stats, including monthly visits trend, time on the site, page-views and bounce rate)

- *Traffic Sources* (a chart with the different sources for a site, including direct links, search, social, mail, display, etc)

- *Geography* (which locations the traffic is coming from; shows 5 leading countries)

- *Referring Sites* (a list of the top 10 inbound and outbound referral sites)

[15] http://www.similarsitesearch.com/

[16] https://chrome.google.com/webstore/detail/similarweb-traffic-rank-w/hoklmmgfnpapgjgcpechhaamimifchmp/

- *Search Traffic* (organic vs. paid traffic and top 10 keywords)
- *Social* (ranks the top 5 social networks according to the quantity of traffic they send to the site)

- *Display Advertising* (top publishers and ad networks, also shows ad screenshots)

- *Audience* (upstream and downstream, other sites that users visited online by category, topic and websites)

- ***Similar sites*** (sites with similar content)

- *Mobile apps* (displays the mobile apps belonging to the website)

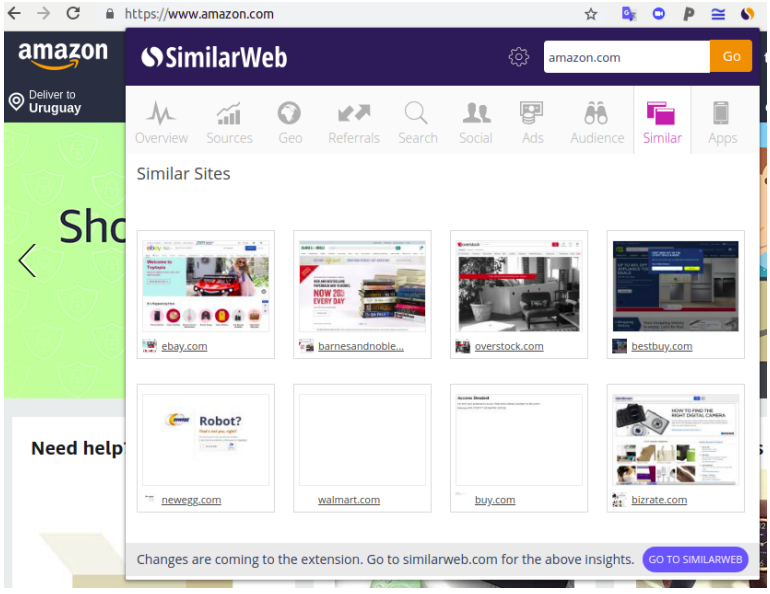


Figure 5 - SimilarWeb extension when browsing www.amazon.com

According to the official documentation[17], the data is gathered from:

- A pool of monitored user devices (hundreds of millions of desktop/mobile devices)

[17] https://www.similarweb.com/ourdata

- Data obtained directly from ISPs
- Web crawlers that scan websites every month
- Direct measurement from websites and mobile apps connected to them

The pool of monitored user devices is the main source of information and it is achieved thanks to the browser extension. According to the extension's agreement and the privacy policy[18], *SimilarWeb* claims to collect data regarding browsing usage, specifically the domains that users browse. The usage of the extension requires granting it permission to capture anonymized *click-stream* data.

Last but not least, the privacy policy also specifies that children under 13 are prohibited from using the service. If *SimilarWeb* becomes aware that a user under the age of 13 has shared any information, the information is discarded. As we will point out later, this restriction makes difficult to learn good similarities for sites where children are the main audience.

2.2.3. Google Similar Pages

Google Similar Pages[19] is an extension specifically designed for the *Chrome* browser by *Google Inc.* As it is shown in *Figure 6*, it displays a few semantically similar pages to the one that the user is browsing.

To the best of our knowledge, *Google* does not provide information about the techniques or algorithms that are used in order to decide what are the most similar pages to a given site.
Probably, not only the data that comes from the browser extension (tracking user's navigation habits) is used for gathering the required information to learn sites similarities. It is reasonable to believe that *Google* also combines information collected from other sources, like the

[18] https://www.similarsites.com/privacy-policy

[19] https://chrome.google.com/webstore/detail/google-similar-pages/pjnfggphgdjblhfjaphkjhfpiiekbbej

contents in themselves (the reader should remember that Google's search engine scans websites for indexing the web periodically) and also from their own public *DNS*, which is a free, global *DNS* resolution service that users can use as an alternative to their current *DNS provider* (this could be much closer to our approach).
Learn semantic information from the contents in themselves is probably the obvious and most direct option but as we will point out later, it is very difficult to implement in large scale systems like the web.

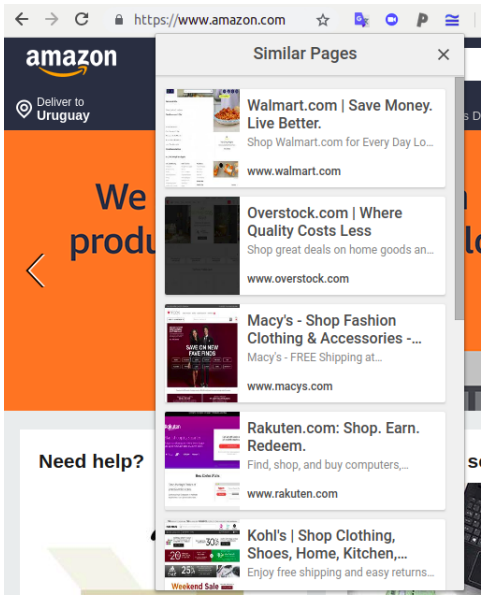


Figure 6 - Google Similar Pages extension when browsing www.amazon.com

2.2.4. Others

There are some other available options that can provide a similar feature for finding similar sites. A brief (non exhaustive) summary with some of these options is presented in the following.

Similarsitesearch[20] is a search engine for finding similar, related or alternative sites. At the moment of writing this, the database is quite small, containing less than 14k items. They use machine learning algorithms and social data to determine the topics of websites which are used to find

[20] https://www.similarsitesearch.com/

similar websites that have the closest matching set of topics. According to this service, multiple aspects are analyzed, including popularity, language, and country of interests. Users can also suggest similar sites for a given domain name (adding a collaborative mechanism for increasing the database). Websites' owners can submit their sites immediately to the database by filling a form and by uploading a file *(ssstopics.txt)* to a webserver in the submitted domain name (for example, http://(www.)yourdomain.com/ssstopics.txt) indicating the topics contained in the site.
This is similar to the keywords tag used in HTML to help search engines to understand the website's content. The platform gives a good explanation of results, saying why a result is similar to a given site. This is a favor point for this tool, since many times when machine learning algorithms are used, the explanations for the results are omitted or they are difficult to interpret.

Moreofit[21] is other search engine for similar sites. Given a website, it suggests alternative highly related and popular websites to explore. Although the service does not provide too much details about how does it work, the site's description claims that they combine manual work for organizing and describing sites with clustering techniques. The manual work clearly is a disadvantage and it is not scalable. This limit the websites considered by the service to just the most popular global domains.
When testing this tool using many of the most popular uruguayan websites, the result was a poor message saying that the provided url is not popular enough to get result.

A topic based directory where sites are submitted manually by users and then grouped under different categories is the approach offered by *SitesLike*[22]*.* For linking similar sites together, the service makes use of known and readily available information like global reach, page rank, keywords and user input.

Top Similar Sites[23] is other possibility, it's a website that offers a basic kind of grouping for some general categories (search, news, social, video, shopping, etc) and a search engine tool for finding similar sites. It also allows to see results order by sites' popularities and see the most

[21] http://www.moreofit.com/
[22] https://siteslike.com/
[23] http://www.topsimilarsites.com/

relevant topics for the site that is being searched. In addition to this, it offers an interesting feature that returns the most similar sites but based on names' syntax and structure.
As we will see later in this work, when proposing a new approach for finding similarities between domain names, morphological characteristics of domain names is also something very important to be considered, since they generally carry strong information about its nature.

Finally, the last option considered in our list is *SimilarSites*[24] that despite being presented like something independent, it is a service powered by *SimilarWeb* (see Section 2.2.2), and the results obtained when looking for similar sites are exactly the same than the results obtained when using *SimilarWeb*.
This service can be used from its website itself, or through its chrome extension[25].
Probably, presenting this service with a different website than *SimilarWeb* (removing all the web analytics tools provided by *SimilarWeb*) is a business strategy decision with the goal of reaching a different kind of public. *SimilarWeb* is thought more for the enterprise and websites' owners who want to get all kind of statistics and competitive information about websites, and on the other hand, *SimilarSites* is thought more for a common Internet user that just want to know similar pages to explore and discover new content similar or related to its interests (it works like a recommender system by recommending similar content to the one that it's being viewed).
The recommendation approach behind *SimilarSites* is *"people who visit this site also like to visit these other related sites"* (just like with Youtube recommendations).
Installing and using *SimilarSites* extension requires granting permissions to capture anonymized browsing data. This powers the algorithms that generate similar sites to the ones a user visits and allows to understand website traffic numbers and flows, which can be used for market research. The same privacy policy[26] than *SimilarWeb* is applied.

[24] http://similarsites.com/
[25] https://chrome.google.com/webstore/detail/similar-sites-discover-re/necpbmbhhdiplmfhmjicabdeighkndkn
[26] https://www.similarsites.com/privacy-policy

2.2.5. Disadvantages of the current approaches: where to go next?

In general we can divide all the previous tools that were presented in two different categories, either those tools that in some way or another require that Internet's users or websites' owners install some kind of component able to run at the client side (a browser plugin/extension, a client side script, etc) or those tools that create a centralized database of similar sites based on sites' topics (manually or automated indexed).
One advantage of many of the previously mentioned approaches is that they can act to the whole URL level, not just the domain. This could be something helpful in different scenarios for example to block only some particular pages of a domain or to recommend specific categories in a domain with multiple subcategories of different products. But as we will see in this section, the current solutions suffer from many disadvantages that motivates the exploration of new approaches like the one presented in this work.

Using client side components has many disadvantages:

- Firstly, it will collect information only from who installed the plugin. This could reduce the public to some specific segment/kind of user, and it is not representative of all Internet users (this is the same problem than a political poll by telephone could have by collecting responses just from people who have telephone, and without considering the opinion of a different kind of voter, the one that cannot have access to a telephone).

- There are legal prohibitions to collect information for children under 13 years old. This restricts the audience even more, making difficult to collect enough big data for sites where children are the main audience. Example of these sites could be educational sites, games sites, cartoons channel websites, among others.

- It requires the user to give permissions to allow the execution of a browser extension (many users do not like this because of security concerns) or allow cross-domain requests from javascript code.

- It is intrusive in the user's navigation and possibly affecting the navigation's performance, and thus, the user experience when visiting different websites.

- It does not work well for mobile devices (not all mobile browsers support using plugins/extensions). This issue, reduce even more the group of users that are tracked. It's important to note that nowadays, mobile access is greater than desktop access [16], so there is a huge segment of users browsing the web from their mobile devices.

- Development, maintenance and updates are difficult. Different extensions for different browsers need to be developed. Users with more than one browser would require to install a different instance of the extension in all browsers. It's difficult to change the centralized service that collect the information once the extensions/site scripts are distributed globally, needing to support possible many different versions at the same time.

- It's difficult to distribute the client side component. Common users probably won't access to download/install the extension explicitly. Important marketing campaigns and ads in popular sites are probably needed to reach a big amount of users, making this approach to be expensive.

- In case of tracking by using scripts hosted in websites, probably only some enterprise sites would use them, and on the other hand, it requires to provide a high quality analytic tool able to provide helpful information for business decisions, and so, to be a sexy option for sites' owners (otherwise, why should they add an script to send information to an external site?)

Using a database of similar sites based on sites' topics presents other disadvantages:

- A high quality service requires a curated database and this is not generally done by an automated process (it requires many people manually checking contents).

- It is based exclusively in the kind of topic a site contains, making difficult to understand complex relationships like complementary sites.

- It requires a submit process to add less popular sites (sites' owners need to submit their websites to the database for an initial tracking).

- Topics are discovered either manually during a curation process or automatically by robots (web crawlers) searching for specific metadata (html keywords, specific text files that describe the site's content, etc). In the last case, it requires sites' owners to add metadata to their sites and we need to trust in their descriptions.

- When similarities are based on metadata, language could be a problem (for instance, keywords in spanish will not match english keywords despite the sites could be very related based on the kind of content they provide).

- When similarities are based on metadata, webmasters could employ different terms for a same concept.

- No matter if the topics are discovered manually or automatically through web crawlers, It does not scale well to all the web.

Other approaches

In the particular case of *Google Similar Pages*, as it was mentioned before, *Google* does not provide information about the techniques or algorithms they use, but we can think that they probably combine information gathered from many sources (not only from the browser extension) like the contents in themselves or their own free public *DNS* resolution service (among others).
Contents could be accessed by *Google* through their search engine, and the similarity problem can be addressed as a classic *Information Retrieval (IR)* problem by creating a vector representation for each content and then just applying some similarity distance between vectors like for example the euclidean or cosine distance (in the next chapter, we will see different

techniques for creating such vector representations for words and documents). But although it sounds great to use the content's information as a basic source of semantic learning, this is not something easy to implement in large scale systems like the web, and it presents lot of disadvantages as an standard solution, being very time consuming and requiring an enormous infrastructure and complex logic to analyze all the contents from the entire web.
On the other hand, if *Google* had been using its own *Google Public DNS System* as a source of information for understanding navigation patterns in the web*,* then it would be a much closer approach than the one presented later in this work. And the same for *SimilarWeb,* which also indicates that they make use of some kind of information provided by *ISPs* but without indicating too much details about this data source.

Hence, since this information is not provided by *Google or SimilarWeb*, then we think that the approach presented in Chapter 4 is a novel method for solving this complex task of finding semantic similarities between Internet domain names.
But before moving to the proposed solution, let's continue presenting the theory behind word and document embeddings, that at the end they are the core ideas used later in this work.

3. Chapter Ⅲ - Vector representation of words and documents

In order to work with different algorithms, computers need to represent words and documents as fixed length vectors that can be used as input for training different machine learning models or for searching relevant documents using some *Information Retrieval System (IRS)*. For example, consider the following two sentences (widely used in the literature)

A. *“Obama speaks to the media in Illinois”*
B. *“The president greets the press in Chicago”*

Probably, we pretend that if we search the sentence *A* in a search engine, then document *B* is included in the results. In the same way, we would like that a machine learning algorithm in the *Natural Language Processing* (*NLP)* field could understand that both sentences have a similar meaning.
In order achieve these goals, computer algorithms need to understand that *“Obama”* and *“president”* have a similar meaning and that they can be interchanged in many contexts as the same thing, and the same with the words *“press”* and *“media”.* Also, intelligent algorithms need to understand more complex relationships between words like the one that exists between *“Illinois”* and *“Chicago”* that despite not being the same thing, they have a strong relationship since Chicago is a city that belongs to the state of Illinois in the United States.

By analyzing big amount of text data (corpora), and how different words are used in different contexts and how they appear combined with other words, different techniques in the *NLP* and *IR* field can be employed in order to build a vector space model *(VSM)*. As we will see later in this chapter, in a *VSM* words are represented as fixed length vectors and the relationships between words can be expressed as mathematical operations between vectors.
These vector representations, have been used for over 50 years, being the most common way to compute semantic similarity between words,

sentences or documents, making these methods an important tool in practical applications like *question answering*, *summarization*, or *automatic essay grading* [17].

A common taxonomy for vector representation of words (a.k.a *word embeddings*) organizes the techniques into two main categories: *count-based* (a.k.a *matrix-factorization based*) and *prediction-based* ([18], [19], [20]). There are many research works like [18], [21], [22], [23], [24], [25] (among others) that describe the main methods commonly used.
In the rest of this chapter we will present a review of these techniques, starting from the most basics ones that use sparse representations, to the most complex ones that are able to learn dense vectors representation using neural networks as part of a statistical language modelling. Statistical models of natural language are a key part of many systems today. The most widely known applications are automatic speech recognition *(ASR)*, machine translation *(MT)* and optical character recognition *(OCR)* [26]. In [27], [28] and [29] a complete review of statistical language models based on neural network can be found.

Later, in Chapter 4 this background knowledge (in particular the word embeddings techniques based on neural networks) will be used to build a vector space model for Internet domain names. In this vector space model, names of websites (*example1.com, example2.com, etc*) are taken from real anonymized *DNS* log queries (user's navigation traces) from a large Internet Service Provider *(ISP)*. The main goal will be to find semantically similar domains only using information of *DNS* queries without any other previous knowledge about the content of those domains.

3.1. One-Hot encoding

This is probably the most simple technique for word representations. Suppose a vocabulary with n words $V = \{w_1, w_2, \ldots, w_n\}$, then the idea behind this technique is to represent each word w_i as a fixed length vector of length n (where n is the number of words in the vocabulary V). Each dimension of the vector $v = (v_1, \ldots, v_n)$ will be zero, except the element at the i^{th} position that will be one.

As an example, if our vocabulary is defined by the words {dog, cat, fish} then, each of these words is represented by a 3 dimensional vector. And the particular encoding for each word would be: 100 (dog), 010 (cat) and 001 (fish).

It is easy to see the main problem of this technique, the high dimensional and sparse representation of the words. This model presents a terrible performance when using large vocabularies, being computational very expensive and making to fail some matrix operations (like rebuild in autoencoders). Furthermore, a representation like this does not include any information about semantic relationship between words.

3.2. Vector Space Models (VSMs)

In the previous example, when using *one-hot embedding*, we pointed out that one of the main issues presented by the method is that the representation does not include any semantic about words.

Vector Space Models (originally introduced in the *SMART* information retrieval system [30] by *Gerard Salton* and his colleagues [31]) on the contrary, allow to represent words by using vectors that when are embedded in a vector space, words with similar semantic are mapped to nearby points in the high dimensional space. From now, and during this work, anytime we refer to a word, sentence or document embedding we are referring to the vector representation for that word, sentence or document in the high dimensional vector space model.

In some way or another, all these vector space models have the notion of *context*, and they have as fundamental hypothesis that similar contexts tend to have similar meanings. The meaning of a word is thus related to the distribution of words around it [17], (for this reason these methods sometimes are called *distributional methods*) and so, words that occur in similar contexts tend to have similar meanings [32][33][34][35]. The different approaches that leverage this principle can be divided into two categories: *count-based methods* and *predictive methods*.

Count-based methods generally use a *co-occurrence matrix*, an structure that allows to define a math model for representing how often words co-occur in a large text corpus and then map these count-statistics down to a small, dense vector for each word [36]. As it was pointed out in [21], the different structures of this co-occurrence matrix (*term– document, word–context, pair–pattern matrices, etc*) are very important in determining the potential applications and can be helpful for organizing the literature on *VSM*s.

Predictive models directly try to predict a word from its neighbors [36] (or the neighbors from a word) and the prediction process can be used to learn embeddings for each target word by starting with a random vector and then iteratively updating the embedding in such way that the embedding is more like the embeddings of neighbor words and less like the embeddings of words that don't occur nearby.

Vector or distributional models of meanings, have been used in the *NLP* field for over 50 years, being the most common way to compute semantic similarity between words, sentences or documents, making these methods an important tool in practical applications like question answering, summarization, or automatic essay grading [17].

In what follows, the most important *VSM* methods that are commonly used are presented, starting from the simplest and most intuitive *count-based* method that leverage the *co-occurrence matrix* in its raw definition (sparse matrix), then moving on to more sophisticated methods that transform the sparse matrix in a dense one, and finally some of the latest approaches currently being used for *predictive models* based on *neural networks* architectures. Later, in next sections, we will focus specifically on the details for the particular *predictive models* that were used through this research.

3.2.1. Count-based methods

3.2.1.1. Sparse Vector representations

3.2.1.1.1. Term-Document Matrix

The *Term-Document Matrix* is probably the simplest and most intuitive *count-based* method that uses a matrix to represent words and documents in a continuous vector space model. In its most basic form, this matrix contains a row for each term in the vocabulary (generally tens of thousands words) and a column for each document in the collection of all possible documents (the collection of documents could be potentially enormous, just think in Internet, for instance). Then, each cell in *matrix(i,j)* represents the number of occurrences that the term *i* occurs in the document *j*.

Although this is a basic and simple structure, it is able to capture semantic from terms and documents which is helpful for applying similarity measures and computations for finding similar terms and documents. For this reason, this approach was initially employed in the Information Retrieval (IR) field, like in [30].

As an example, consider a vocabulary $V=\{w_1, w_2, w_3, w_4, w_5, w_6\}$ that corresponds to the union of all the words that are present in documents d_1, d_2, d_3. Suppose that the text content (corpus) of these documents are:

- $d_1 = w_1\ w_2\ w_1\ w_3$
- $d_2 = w_2\ w_3\ w_2\ w_3\ w_4$
- $d_3 = w_3\ w_1\ w_5\ w_6\ w_4\ w_5\ w_6\ w_5\ w_4\ w_1$

Then, the *Term-Document Matrix* is:

	d_1	d_2	d_3
w_1	2	0	2
w_2	1	2	0
w_3	1	2	1
w_4	0	1	2
w_5	0	0	3
w_6	0	0	2

Table 3 - Term-Document Matrix example

Now, the columns and rows can be used as vectors to represent documents and words respectively in the vector space model. The column representation for documents, is also known as *bag of words* representation, since the semantic of the document is defined exclusively by the set of words that appear in the document (order doesn't matter). For instance, $v_{d1}=(2,1,1,0,0,0)$ is the 6 dimensional vector that represent the document d_1, $v_{d2}=(0,2,2,1,0,0)$ and $v_{d3}=(2,0,1,2,3,2)$ are the vectors that represent the documents d_2 and d_3 respectively. At the row level we have the 3 dimensional vectors $v_{w1}=(2,0,2)$, $v_{w2}=(1,2,0)$, $v_{w3}=(1,2,1)$, $v_{w4}=(0,1,2)$, $v_{w5}=(0,0,3)$ and $v_{w6}=(0,0,2)$ that represent the words w_1, w_2, w_3, w_4, w_5 ,w_6 respectively.

It is interesting to note that words and documents live in different dimensional spaces. For modeling the documents d_1, d_2 and d_3, we have a 6 dimensional space, where each dimension corresponds to the number of occurrences of an specific term in the document. For modeling terms w_1, w_2, w_3, w_4, w_5, w_6 we have a 3 dimensional space, where each dimension corresponds to the number of occurrences that the term has in an specific document. If we generalize this observation, for an $N \times M$ matrix where $N = |V|$ (number of unique words in the vocabulary) and $M = |D|$ (number of documents in the documents collection) the vector space model for the terms is a M dimensional space and the vector space model for modeling documents is a N dimensional space.

Finally, since similar documents use similar terms that define the semantic of the document, those documents will have similar values in the dimensions that correspond to those terms. Then, after having found a vector representation of the documents that keeps some of the hidden semantic, we can compute similarity between documents by measuring the distance between the vector representations of the documents.
And the same idea also applies to compute similarity between words. Since similar words tends to appears in the same documents, then their vector representations tends to have similar values in the dimension that corresponds to those documents, and so, the distance between those words tends to be small.

One of the most common distance metric widely used is the cosine distance. Cosine distance between two vectors is defined in [21] by *Equation 1*.

$$\text{cosine}(\vec{v},\vec{w}) = \frac{\vec{v}\cdot\vec{w}}{|\vec{v}||\vec{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}} \quad .$$

(*Eq. 1*)

For this metric, the most important is the angle between the vectors and not the length of the vectors. When the two vectors point in opposite directions (180 degrees) then the cosine value is -1, when they point in same direction (0 degree) the cosine value is 1, and when they are orthogonal (90 degrees) the cosine value is 0.
Having presented the cosine distance formula, we can use it to measure how similar two words or two documents are by considering their vector representations from the *term-document* matrix. For instance, the cosine distance between w_1 and w_2 is 0.316, between w_1 and w_3 is 0.577, between d_1 and d_2 is 0.544 and between d_1 and d_3 is 0.435. By using these metrics, we can see easily that w_1 and w_3 are more similar than w_1 and w_2, and d_1 is more similar to d_2 than d_3.

One disadvantage is that when using raw frequencies of word occurrences in documents, common words that appear in all documents and do not add any important semantic (like english stop-words: is, the, a, an, it, etc) to the document, can introduce noise in the representation when computing similarities.
As we are about to see, weighting techniques like *PMI (Pointwise Mutual Information)* and the *tf-idf (term frequency – Inverse document frequency)* are commonly used to avoid the issue of common terms that are not as relevant as others less frequent terms, and can affect the performance of the similarity metric.

Finally, it's important to mention that other two disadvantages of the *term-document* matrix are that rows and columns vectors that represent words and documents respectively, generally have a high number of dimensions (requiring lot of space for allocating the whole matrix) and they are also very sparse (most values in vectors are zero).
As we are going to see later, dense vector representations of words and documents will help to solve these problems.

3.2.1.1.2. Weighted Term-Document Matrix

Using the raw frequency of terms in documents as values for the *Term-Document* matrix could be not the best idea. This is because not all words in a document are equally important [37]. For example, common words like "*the", "and", "this", "it*" (among others) can appear tons of times in each document and also are words that probably occurs in all documents, so they do not add any important meaning to the documents where they appear.

Common words, can add noise when calculating the similarity between documents or words. One option to deal with this problem is to remove the more frequents words or words that appear in some predefined blacklist or stopwords list. Anyway, and although removing stop words can help removing noise for computing similarities, we still can have words that are more frequent than others adding less semantic than those words that are very specific for particular domains and so carrying lot of semantic information in it.

In order to deal with this problem, a common approach is to use weighted values for term frequencies in documents instead of using just raw frequencies. A weighted value can be seen as the output of some function of relevance or importance of the term, applied to the raw frequency value of that term.
This approach has demonstrated to work very well and different weighting functions have been used. Two of the most well-known are *PMI (pointwise mutual information)* and *tf-idf (term frequency – Inverse document frequency)*

3.2.1.1.2.1. Pointwise Mutual Information (PMI)

For computing word similarity, PMI is one of the most popular techniques employed for weighting term frequencies in a *Term-Context* matrix (as we will see, the Term-Context matrix is a generalization of the Term-Document matrix, allowing different kind of contexts and not only documents).

After applying PMI to the original matrix, then each cell in the final matrix represents the association between the word w_i and the context c_j . We can think about PMI as a measure of how often two words co-occur in the same context, compared with what we would expect if they were independent [17].

More formally, when computing PMI between two words *x* and *y*, PMI *"compares the probability of observing x and y together (the joint probability) with the probabilities of observing x and y independently (chance)"* [38].
[38] also presents *Equation 2* as the formula for estimating PMI as the log ratio between x and y's joint probability and the product of their marginal probabilities.

$$PMI(x,y) \equiv log\frac{p(x,y)}{p(x)p(y)} \quad , \qquad \text{(Eq. 2)}$$

where:

- *P(x)* is the estimated probability of the word x. In practice, we represent this probability as the number of occurrences of *x* in the corpus
- *P(y)* is the estimated probability of the word y. In practice, we represent this probability as the number of occurrences of y in the corpus
- *P(x, y)* is the estimated probability that words x and y occur in the same context. In practice, we represent this probability as the number of times where the two words co-occur in the same context

Furthermore, a minor variation of PMI called *Positive Pointwise Mutual Information (PPMI)* is typically used. As shown in *Equation 3*, PPMI simply corrects values lower than zero to zero.

$$PPMI(x,y) = max(PMI(x,y),0) \ . \qquad \text{(Eq. 3)}$$

In that way, PPMI is never negative and according to [39] it reaches a better performance than PMI.

Regarding to the disadvantages of PMI, a well-known problem is that it is biased towards infrequent events. [40] and [41] provide some approaches to deal with this problem.

As a final note, it is important to highlight that PMI works well with both the Term-Context matrix and also with the Term-Document matrix [40]

3.2.1.1.2.2. Term frequency - Inverse document frequency (TF-IDF)

This technique originally developed in the Information retrieval (IR) field is a very popular technique for weighting values in the Term-Document matrix. The main idea behind this technique is to consider the number of documents where a term appears as a measure of how common the term is. Then, if you have two different terms T_x and T_y with the same number of occurrences in a document *D*, the output value given

to T_x will be higher than the output value given to T_y if and only if the number of documents containing T_x is lower than the number of documents containing T_y.

In order to formalize this idea, given a term *t*, the *Inverse Document Frequency (IDF)* definition comes into play, and it is defined by *Equation 4* as follows:

$$idf_t = log(N/df_t) \ , \qquad \text{(Eq. 4)}$$

where *N* is the total number of documents and df_t is the number of documents where the term *t* appears. Finally, as shown in *Equation 5* the *TF-IDF* formula is just the product of the raw *term frequency (TF)* of the term *t* in the document *d* and its *IDF* weight.

$$tfidf_{t,d} = tf_{t,d} \cdot idf_t \ . \qquad \text{(Eq. 5)}$$

Equation 5 contains all the desired characteristics, assigning to term *t* a weight in document *d* that is [37]:

1. highest when *t* occurs many times within a small number of documents (thus lending high discriminating power to those documents)

2. lower when the term occurs fewer times in a document, or occurs in many documents (thus offering a less pronounced relevance signal)

3. lowest when the term occurs in virtually all documents

Tf-idf thus prefers words that are frequent in the current document *d* but rare overall in the collection. The *tf-idf* is by far the dominant way of weighting co-occurrence matrices in information retrieval, however, is not as common as *PPMI* as a component in measures of word similarity [17].

3.2.1.1.3. Term-Context Matrix

This is a generalization of the Term-Document Matrix, allowing different contexts instead of only documents. Some of the most common options are phrases, sentences, paragraphs, chapters, documents, or more exotic possibilities, such as sequences of characters or patterns [21]. Sometimes this matrix is also called the *term-term matrix* or the *word-word matrix* because rows and columns contain all the terms (words) of the vocabulary and each cell in this |*V*| x |*V*| matrix allocates the number of co-occurrences of the two words in same contexts (number of shared contexts between the two words).
Although many kind of contexts can be used (like the Term-Document matrix where the contexts are entire documents) generally, small contexts are used, for instance 4 words before and after a central word.

As an illustrative example, using the same words and documents than in the previous example, and considering a small window of size 1, then the term-term matrix can be expressed as shown in *Table 4*.

When using small window sizes, since the information is coming from immediately nearby words, the representations tends to be more syntactic. On the contrary, when longer the window, the more semantic the relations [17].

	w1	***w2***	***w3***	***w4***	***w5***	***w6***
w1	1	3	4	2	3	1
w2	3	1	5	1	0	0
w3	4	5	1	2	1	0
w4	2	1	2	0	5	4
w5	3	0	1	5	1	6
w6	1	0	0	4	6	0

Table 4 - Term-Term Matrix example

As a good example of this kind of approaches we have the *Hyperspace Analogue to Language (HAL)*[42], that creates a word-word co-occurrence matrix where rows and columns represent words and cells contains the number of times a given word (row) occurs in the context of another given word (column).
One of the majors problems with *HAL* and related methods is that the most frequent words like *the*, *or* or *and* (among others) contribute a disproportionate amount to the similarity measure, having a large effect on their similarity despite conveying relatively little about their semantic relatedness [19].

Techniques like *COALS* method[43] or the square root type transformation in the form of *Hellinger PCA (HPCA)*[44] have demonstrated to be helpful to address this issue, by initially transform the co-occurrence matrix using an entropy or correlation-based normalization [19].

Finally, it's worth mentioning that other disadvantage of the *Term-Context Matrix* is that it suffers the issue of being high dimensional and sparse (most of the entries are zero) as we already observed for the *Term-Document matrix*. In particular, this problem is even more visible here because the matrix size in this case is $|V| \times |V|$ where V represents the vocabulary set. Lower dimensional and dense vector representations will help to solve this problem as we will see in the following sections.

3.2.1.2. Dense Vector representations

When studying the *Term-Context matrix* representation for embedding words we pointed out that the two main disadvantages were that the vectors were very long (each word represented by a vector with one dimension for each word in the vocabulary) and also very sparse (most of the vector's entries were zero).
These two disadvantages are the issues that dense vector representations try to address. In other words, the goal of a dense vector representation is to represent words by capturing their semantic in a shorter and dense vector (generally from 50 to 1000 dimensions of real numbers without zeros)

In what follow we present some of the most popular count-based methods for learning low dimensional and dense representation of words. Later we will see that also other techniques based in neural networks and predictive models can be used with the same goal.

3.2.1.2.1. Latent Semantic Analysis (LSA)

This technique, sometimes also referred as *Latent Semantic Indexing (LSI)* is maybe the most intuitive evolution from the long and sparse representation given by the *Term-Document matrix* approach.
After building the *Term-Document matrix*, *LSA* simply apply a dimensional reduction approach like the *Singular Value Decomposition (SVD)* technique to generate a lower dimensional and dense representation of the original *Term-Document matrix*.

It's worth mentioning that a weighting function is applied to the *Term-Document entries* in the matrix, instead of using just the raw frequencies. *LSA* applies both, a local and global weighting function to each nonzero element, in order to increase or decrease the importance of types within documents (local) and across the entire document collection (global).
The local and global weighting functions for each element, are usually directly related to how frequently a type occurs within a document and inversely related to how frequently a type occurs in documents across the collection, respectively [45].
In Section 3.2.1.1.2 some of the most common approaches for creating a weighted Term-Document Matrix were already introduced. Also the reading of [46] is a good reference for a more detailed explanation about local and global weighting functions.

Now, in order to understand how *LSA* works, suppose a dataset with *c* documents and a vocabulary *V*. If *X* represents the *Term-Document matrix*, then after applying *SVD* we can find a matrix factorization such like the one shown in *Equation 6*.

We can think this factorization as a way of adding to the equation the *m* hidden latent factors that are more important for the specific set of

words and documents and then rewrite *X* in terms of those *m* factors [47] [48].

$$\underset{|V|\times c}{\begin{bmatrix} X \end{bmatrix}} = \underset{|V|\times m}{\begin{bmatrix} W \end{bmatrix}} \underset{m\times m}{\begin{bmatrix} \sigma_1 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \sigma_m \end{bmatrix}} \underset{m\times c}{\begin{bmatrix} C \end{bmatrix}} .$$

(*Eq. 6*)

After applying this factorization, the *sigma matrix* contains values in its diagonal that indicate how important each hidden factor is (sigma is a diagonal matrix of size *m x m* with singular values in decreasing order).

LSA leverages this information provided by the *sigma matrix*, and takes the top *k* most relevant dimensions (*k* is a parameter for the model) that capture the most important features with the highest variance in the data and then it approximates the original *Term-Term matrix* as shown in *Equation 7*.

Finally, each of the |*V*| rows of the *W* matrix can be used as a small (*k* is taken much lower than *m*, generally near to 300) dense vector representation for each word in the vocabulary *V*. Similarly, columns of the *C* matrix can be used as dense representations for documents.

$$\underset{|V|\times c}{\begin{bmatrix} X \end{bmatrix}} = \underset{|V|\times k}{\begin{bmatrix} W_k \end{bmatrix}} \underset{k\times k}{\begin{bmatrix} \sigma_1 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \sigma_k \end{bmatrix}} \underset{k\times c}{\begin{bmatrix} C \end{bmatrix}} .$$

(*Eq. 7*)

The cost of executing the dimensional reduction process is very significant, but generally (not always) the dimensional reduction applied by *SVD* ends up by generating a model that performs better than the raw *Term-Document matrix*. When this occurs, is typically because removing low important dimensions allows to generalize better and avoid the overfitting issue [48] [49].

3.2.1.2.2. SVD applied to the Term-Term Matrix

This approach is very similar to *LSA*, but in this case, *SVD* is applied to the *Term-Term matrix* instead of the *Term-Document matrix*.
The typical size of *k* used for truncating the matrix *W* that composes the matrix factorization after applying *SVD* is between 500 and 5000, instead the 300 dimensions typically used by *LSA*. This difference is because of the granularity difference between the contexts, while the context employed by *LSA* are whole documents, the context used in *Term-Term matrix* are small windows of neighboring words [17].

In regards to the disadvantages, the same issue already explained about performance and cost of execution of the dimensional reduction process of the *SVD* technique is also valid here.

Next in this work, we will present other kind of dense vector representation approach based on the usage of *Neural Networks* that will address the significant computation cost of the dimensional reduction process applied to the *Term-Context matrix* when computing dense vectors with a count-based method.

3.2.2. Prediction-based Models

In this section, we will focus on a different approach for finding vector representation of words. Inspired by probabilistic language models, we will introduce some of the most important techniques for learning high quality representation of words based on words predictions.

Firstly, one of the simplest and powerful approaches for language modelling based on probabilities is described: the *N-gram* model. Although *N-grams* models are not used directly as a word embedding technique, it gives us the basis of a simple predictive model, being important in order to understand the evolution of language predictive models. Then, an evolution of the *N-gram* model is presented by introducing artificial neural networks as an artifact for improving the task of predicting the next word given a sequence of previous words. In this category, *Neural Network Language Models (NNLM)* are studied, in particular, the *Recurrent Neural Network Language Models (RNNLM)* and the feedforward architecture proposed in [50]. This last one ([50]), has became a reference work, and it has been the inspiration for many other researches and works like the *Skip-Gram* and *CBOW* architectures used in *Word2Vec*, *App2Vec* and *FastText*, that are also presented later in this section. The study of these models has particular interest, since they are used and compared later in this work as part of the main research proposed in this thesis.

3.2.2.1. N-gram model

Despite *N-gram* is probably the simplest language model *(LM)* that we can find, it is also one of the most widely used tool in language processing. As any other *LM* it assigns probabilities to sequence of words, and it allows to estimate the probability of the next word in a phrase given the previous words.

Given the history $h=(w_1, w_2, \ldots, w_{n-1})$ of the previous words before w_n then, the joint probability $P(w_1, w_2, \ldots, w_{n-1}, w_n)$ can be calculated using the *chain rule of probability*[27] as:

$$P(w_1, w_2, \ldots, w_{n-1}, w_n) = P(w_n | w_1, w_2, \ldots, w_{n-1}) \cdot P(w_1, w_2, \ldots, w_{n-1}) =$$
$$P(w_n | w_1, w_2, \ldots, w_{n-1}) \cdot P(w_{n-1} | w_1, w_2, \ldots,) \cdot \ldots \cdot P(w_1, w_2, \ldots.) =$$
$$P(w_n | w_1, w_2, \ldots, w_{n-1}) \cdot P(w_{n-1} | w_1, w_2, \ldots,) \cdot \ldots \cdot P(w_2, | w_1) \cdot P(w_1)$$

which can be reduced to *Equation 8*.

[27] https://www.ibm.com/developerworks/community/blogs/nlp/entry/the_chain_rule_of_probability?lang=en

$$P(w_1, w_2, .., w_n) = \prod_{K=1}^{n} P(w_k | w_1, .., w_{k-1}) \quad .$$

(*Eq. 8*)

As we can see, *Equation 8* needs to compute the conditional probability of a word given its full previous history. These probabilities are based on the number of occurrences of the entire sequence of length *n* compared to all existing sequences of length *n* (that share the same prefix of *n-1* words), and this is impractical in most cases for high values of *n*.
For this reason, *n-grams* uses an approximation of this previous formula, using just the recent history of previous words instead of the full history. Using the recent history of the previous words for modeling language, is supported by *Markov* chains, when bigrams and trigrams were used in [51] to predict whether an upcoming letter in Pushkin's *Eugene Onegin* would be a vowel or a consonant.
Markov chains assume that we can predict the probability of some future unit without looking too far into the past [17]. This simple yet powerful hypothesis, is one of the fundamentals that other evolutions of language models will take, like the concept of word's context used by predictive neural network models, as we will see in next sections.

The *n-gram* that considers just the most recent previous word is called *2-gram* (or bigram), the one that considers the last two previous words is called *3-gram* (or trigram), and so on. Hence, an *m-gram* can calculate the probability of a word w_n given its previous *m-1* words using the approximation shown in *Equation 9*.

$$P(w_1, w_2, .., w_n) \approx \prod_{k=n-m+1}^{n} P(w_k | w_{n-m+1}, .., w_{k-1}) \quad .$$

(*Eq. 9*)

In order to calculate these probabilities different techniques can be used. One of the most intuitive ways of calculating them is using what is called as *Maximum Likelihood Estimation (MLE),* where conditional probabilities at the right side in *Equation 9* can be calculated as:

$$P(w_k | w_{n-m+1}, .., w_{k-1}) = \frac{Count(w_{n-m+1}, .., w_{n-1}, w_n)}{Count(w_{n-m+1}, .., w_{n-1})} \quad ,$$

(*Eq. 10*)

where the numerator represents the number of occurrences of the entire sequence of *m* words (ending with the word w_n) and the denominator represents the number of *all* the possibles occurrences of length *m* that start with the *same* prefix (w_{n-m+1}, … , w_{n-1}) of *m-1* words (the only difference between them is the last word). The effect of this denominator is the normalization of the sequence of *m* words in order to get a probability value between 0 and 1.
Note that if for an specific sequence *S* of *m* words that ends with the word w_n all the possible sequences of *m* words that share the same prefix of *m-1* words with *S* also end with w_n, then the numerator and denominator in this expression are equal, and then, the probability is 1 (meaning that given this same prefix of *m-1* words, we are one hundred percent sure that the next word is w_n).
Also it's easy to see that if the sequence *S* never occurs in the corpus, then the conditional probability that the next word is w_n given that the prefix is (w_{n-m+1}, ... , w_{n-1}) is zero (we are one hundred percent sure that the next word will not be w_n)

But, in a real world scenario, when using *n-grams* for computing probabilities for unseen data, the border cases of assigning probabilities values of 0 or 1 are problematic. If an *n-gram* model is trained on a training corpus where an specific sequence of length *n* never occurs, we can not be sure that in a different corpus (unseen data) this specific sequence will not be present. Similarly, if in a training corpus we see that all occurrences of a same prefix is followed by the same word, we can not be sure that in a different corpus there is no other word that could continue the sequence.

Smoothing algorithms for *n-grams* models like *Backoff* or *Interpolation* try to minimize this problem by providing a more sophisticated way to estimate the probability of *n-grams*, shaving off a bit of probability mass from some more frequent events and give it to the events we've never seen. Also, many others smoothing algorithms like *add-1 smoothing*, *add-k smoothing*, *stupid backoff*, or *Kneser-Ney smoothing* exist, in particular to avoid the zero probability issue. The modified *Kneser-Ney smoothing (KN)* is reported to provide consistently the best results among smoothing techniques, at least for word-based language models [52]. A more detailed description of *Smoothing* algorithms can be found in [17].

But although having many different and complex algorithm is important, empirical studies have repeatedly shown that simple algorithms can often outperform their more complicated counterparts in wide varieties of *NLP* applications with large datasets [53], and many believe that it is the size of data, not the sophistication of the algorithms that ultimately play the central role in modern *NLP* [6].
For this reason, big companies like *Google* or *Microsoft* have released their own *n-grams* datasets. In 2006, *Google* released the *1 Tera-word Google N-gram*[28] [54] and in 2010, *Microsoft* released the *Microsoft web N-gram corpus*[29] [53]. Other big datasets that have been found through this research are the *English Giga-word corpus*[30] [55] and the most recent at the moment of writing this thesis, called *"N-grams data"* [31].

As we have seen in this section about *N-grams* models, the probability computed by this kind of models in order to make predictions is highly based on counting of words, sequences of words, and statistics.
In the next section, we will move to study a different approach, based on the usage of neural networks as a different artifact for training a language model and using the network's weights as good embeddings for representing words.

3.2.2.2. Neural Network Language Models (NNLM)

In the previous section we presented *N-grams* as one simple yet powerful language model, that has been widely used in many *NLP* related tasks, in particular to predict the next word that follows in a sequence (generally a short history of the previous words).

The main problem with *N-grams* and this kind of standard language models is that the number of parameters increases exponentially as the *n-gram* order increases, and *n-grams* have no way to generalize well from training to test set [17] (although as we pointed out, some tricks like *smoothing* exist with the objective of minimizing this problem).

[28] https://catalog.ldc.upenn.edu/products/LDC2006T13

[29] https://blogs.msdn.microsoft.com/webngram/
[30] https://catalog.ldc.upenn.edu/LDC2003T05
[31] https://www.ngrams.info/intro.asp

A predictive language model that has been proposed as an alternative to *N-grams* is based on a different approach: the usage of *Artificial Neural Networks (ANN)*. *ANN* can be used as an artifact to help in the prediction task of a language model. Additionally, as we will see in this section, it allows learning distributed representations of words (embeddings) while adjusting the weights of neurons during the training phase (typically applying *backpropagation* with *stochastic gradient descent*). At the time of this writing, *Neural Network Language Models (NNLMs)* are the kind of language models that have the highest accuracy.

Similar to *N-grams* models, *NNLMs* use the recent previous history of a word as input for the training phase, and that is why typically in the associated literature we find the concepts of word's context or window-based approach when we study this kind of language models. Also, since the prediction task is trained using a word and its context, the learned embeddings have the property of being similar for words that appear in similar contexts but totally different for words that typically do not share similar contexts. As we will see, this property will allow the usage of word embeddings for finding semantically related words or analogy reasoning.
Additionally, since these embeddings are vector representations with a fixed size (typically between 100 and 300), the learned embeddings solve the sparsity issue for word representations when using *count-based* methods like the ones that we studied in Section 3.2.1.1.

In what follows, we will present the most common neural network architectures for language modelling: the *Feedforward Neural Network Language Model* [27] [28] and the *Recurrent Neural Network Language Model* [29].

3.2.2.2.1. Feedforward Neural Language Model (FFNLM)

Feedforward neural language models were introduced in [50]. This kind of language model is a standard feedforward network (with an input layer, one or more hidden layers and an output layer, where the connections between network's neurons go from layer *n* to layer *n+1* and they are generally fully connected) that takes as input at time *t* a representation of some number of previous words (w_{t-1},w_{t-2}, *etc*) and

outputs a probability distribution over possible next words [17]. These previous words (w_{t-1},w_{t-2}, *etc*) are generally called *context* and a common used strategy for representing these words is the one-hot encoding representation that was described in section 3.1.
At this point, the reader probably notices that this trick of using previous words is very similar to the approach used by the *N-grams* models, where the probability of a word w_n given its previous *m-1* words was calculated using *Equation 9*.
In the same way, a *feedforward neural language model* outputs a probability distribution over possible next words considering just the context words as input. That is, given the set of previous recent words, the output contains the probability of each word in the vocabulary to be the next word in the sequence, and the sum of all these probabilities is 1.

If we recall, the *one-hot encoding* strategy used a vector of size |*V*| (being *V* the vocabulary) for representing a word w_i, where all dimensions in the vector are 0 except the dimension *i* that is 1) . Hence if the context length is *L*, the size of the input layer (number of neurons) is *L* x |*V*|.

In regards to the output layer, there is one neuron per word *w* in *V,* hence the size of the output layer is |*V*|. Furthermore, the output of each neuron *i* in the output layer represents the probability of word w_i to be the next word that follows in the sequence (after the input context words). Notice that the task of predicting the next word that follows in the sequence is a classification problem, where there are |*V*| posibles words as candidates (*a.k.a* classes). Generally, for multi-class classification problems a *softmax function*[32] is used as the nonlinear activation function for the output layer. It has the property of generating values between 0 and 1 (a probability) and also it distributes the probabilities among all the classes by applying a normalization, resulting that the sum of all the probabilities for all neurons in the output layer sums 1 (the next word will be w_1 *or* w_2 *or … or* w_n).

In the original proposal in [50], the *tanh (hyperbolic tangent)*[33] nonlinear activation function was chosen for the hidden layer. Additionally, a projection layer was added between the input layer and the hidden layer.

[32] http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-12.html
[33] http://functions.wolfram.com/ElementaryFunctions/Tanh/introductions/Tanh/ShowAll.html

Since the one-hot vectors that are fed into the input layer have a value of 1 in just one of the $|V|$ dimensions, then this projection layer acts just a look-up table to select the current embeddings associated to the input words. The projection layer does not apply any activation function for generating the neuron's output (we may say that it uses an identity or linear function as the activation function, but the relevant thing here is that the projection layer does not use any nonlinear function), and for that reason sometimes in the bibliography it is not considered as part of the hidden layers.

Figure 7 summarizes the high level architecture of the feedforward neural language model presented in [50].

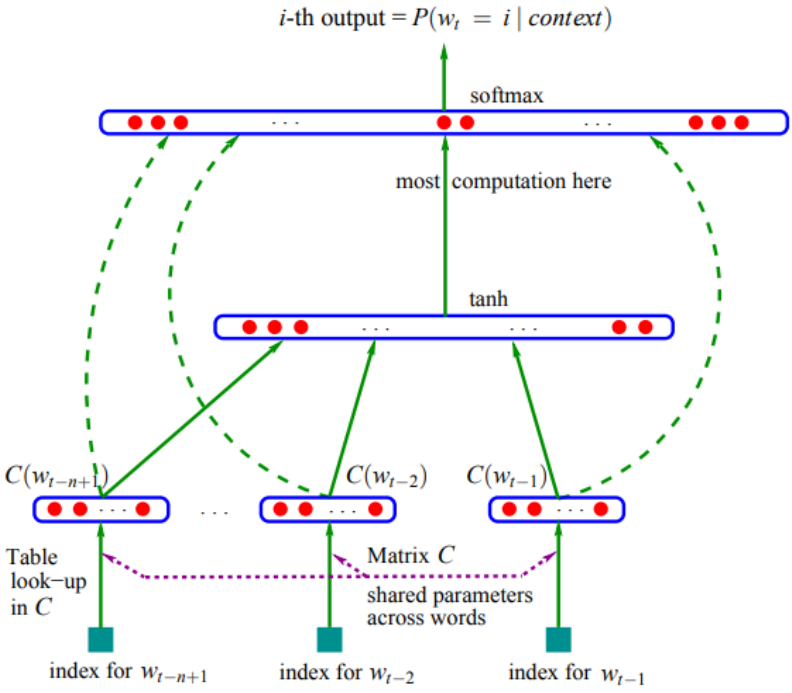


Figure 7 - High level architecture of the feedforward neural language model. Source [50]

In its background, the architecture of this neural network is computing the function $f(i,w_{t-1},\cdots,w_{t-n+1}) = g(i,C(w_{t-1}),\cdots,C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the *i-th* word feature vector [50]. The usage of a linear projection layer and a nonlinear hidden layer in this

architecture, allows to learn jointly the word vector representation and a statistical language model [56] (notice that weights in the *C* matrix actually corresponds to the word embeddings that we are interested in) and it has been the inspiration of many others works.

As it is highlighted in [50] the main problem with this architecture is the complexity associated to the nonlinear hidden layer, resulting in a high computational cost. Additionally an extra cost is associated to the normalization applied by the *softmax* function in the output layer.

Later in this work we will see how simpler approaches like the *Skip-Gram* or *CBOW* architectures employed in *word2vec* take this base architecture and by removing the nonlinearity complexity of the hidden layer and using alternatives to the *softmax* function for the output layer, are able to learn high quality representations (embeddings) of words in a much efficient way.

But before moving to that, let's see other kind of neural network architecture that has been widely used for language models and that is one of the most powerful nowadays: the *recurrent neural networks language models (RNNLM)*.

3.2.2.2.2. Recurrent Neural Network Language Model (RNNLM)

So far we have seen how statistical language models like *N-grams* or those based on neural networks like the *Feed Forward Neural Network Language Model* can be employed to solve the problem of predicting upcoming words based on some previous words (recent previous history). We also have noticed that when using neural network based language models, once the network is trained, the weights of neurons can be used as a good representation of words, and we have been calling to these representations: word embeddings (the main concept behind this work).

Hence, the study of language models is crucial for understanding word embeddings and seeing the historical evolution of these models is important to understand the issues that new models try to solve. For this reason, a new kind of language model is introduced now: *Recurrent Neural Network Language Model (RNNLM)*.

The architecture design behind *Recurrent Neural Networks (RNNs)* allows to *remember* information that has been fed into the network in previous iterations during the training phase. In machine learning, and more specifically in the neural networks field, *RNNs* have been historically used to deal with sequence problems. And, since the main goal of language models is intrinsically related to a sequence learning problem (where we want to predict the next word in a sentence by knowing the previous sequence of words that have occurred in the recent past) is naturally reasonable to think that *RNNs* are good candidates to be used for language modelling.

The original *RNNLM* is described by *Mikolov et al.* in [57] and it is presented as a *"simple"* recurrent neural network (or also called *Elman network* [58]) language model. *Figure 8* shows the high level architecture of the original *RNNLM*:

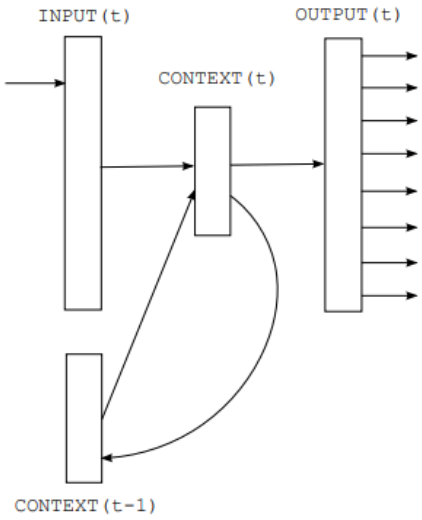


Figure 8- High level architecture of the recurrent neural language model. Source [57]

As it is shown in the high level architecture definition, *RNNLM* does not have a projection layer; only input, hidden and output layer and rather than use a fixed input context for modelling the previous history (like both, *N-grams* and *FFNLM* which use the previous *m-1* words*), RNNLM* uses recurrent time delayed connections (outputs from the hidden/context layer

at time *t-1* are connected to the input layer at time *t*) and by doing this, information can cycle inside these networks for arbitrarily long time (see [59]), allowing the model to learn how to remember from past variable length histories.
As shown in *Equation 11,* in order to form the input *x(t)* for the network at time t, both the representation of the current word *w(t)* and the output from the hidden layer *s(t-1)* at time *t-1* are used.

$$x(t) = w(t) + s(t-1) \ . \qquad \textit{(Eq. 11)}$$

The *plus* sign in *Equation 11* denotes the concatenation operation between the two vectors *w(t)* and *s(t-1),* and the size of *x(t)* is the sum of the sizes of *w(t)* and *s(t-1)*. Like in *FFNLM*, *one-hot encoding* is used to represent *w(t)*, then the size of *w(t)* is |*V*|. The size of *s(t-1)* is equal to the number of neurons in the hidden layer, and this is the only hyper-parameter that needs to be tuned when using a *RNNLM* (this is another advantage over *FFNLM* that needs to specify not only the size of the hidden layer, also the the size of the projection layer and the context's length).

In regards to the activation functions that are used by the original *RNNLM*, a *sigmoid* function (*Equation 12*) is used for the hidden layer (this is also another change compared with *FFNLM* that generally uses a *tangh* activation function for the non-linear hidden layer) and a *softmax* function (*Equation 13*) continues being the option for modeling the probability distribution for the output layer.

$$sigmoid(z) = \frac{1}{1+e^{-z}} \ . \qquad \textit{(Eq. 12)}$$

$$softmax(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}} \ . \qquad \textit{(Eq. 13)}$$

The original *RNNLM* is trained using standard backpropagation with *Stochastic Gradient Descent (SGD)* and convergence is usually achieved after 10-20 epochs [57]. And, although it is often claimed that learning long-term dependencies by *SGD* can be quite difficult [60] *RNNLM* has

shown to significantly outperform many competitive language modeling techniques in terms of accuracy [26] (and also it requires much less data than other models).

In order to improve model's performance, rare words (words that occurs less often than a threshold in the vocabulary) are merged into a single token. Hence, since the size of the output layer *y(t)* is defined by the number of words in the vocabulary, by doing this merge, the size of the output layer *y(t)* can be reduced significantly and then, the complexity associated to computation of the softmax function is reduced too.

Equation 14 summarizes how to compute the probability of the next word $w_i(t+1)$ given the current word *w(t)* and the representation of previous history given by the hidden layer state *s(t-1)*. For rare words, the probability is distributed uniformly between them, and for not rare words, the probability corresponds just to the output value of the neuron $y_i(t)$ in the output layer.

$$P(w_i(t+1)|w(t), s(t-1)) = \begin{cases} \frac{y_{rare}(t)}{C_{rare}} & \text{if } w_i(t+1) \text{ is rare,} \\ y_i(t) & \text{otherwise.} \end{cases} \quad \text{(Eq. 14)}$$

In *Equation 14* $y_{rare}(t)$ is the output value of the neuron associated to the token that represents rare words in the output layer, and C_{rare} is the total number of rare words.

This trick of merging rare words into a single token has proven to improve the model's performance considerably, but unfortunately this improvement is not enough for using *RNNLM* in many real world scenarios with large scale datasets.

Hence, while this model has shown to significantly outperform many competitive language modeling techniques in terms of accuracy, the remaining problem is the computational complexity [26].
For this reason *Mikolov et al.* presented a second work [26] where several modifications to the original *RNNLM* were proposed that lead to more than 15 times speedup for both training and testing phases.
In this extension to the original work, the reduction of the amount of parameters in the model is addressed. Also, a variant to the standard

backpropagation algorithm known as *backpropagation through time (BPTT)* is presented, resulting into a *RNN* model that can be smaller, faster (both during training and testing), and more accurate than the original *RNNLM*.

With truncated *BPTT*, the error is propagated through recurrent connections back in time for a specific number of time steps, allowing the network learns to remember information for several time steps in the hidden layer (see [59]). Better accuracy also is gained by combining *RNN* models linearly (similar to random forests, that are composed of different decision trees) that differ either in random initialization of weights or also in the numbers of parameters. Computational and space complexity is reduced by using classes, factorization of the output layer and compression layers. Experiments showed in [26] demonstrate that the extended model outperforms the original one, presenting significant improvements when comparing the models using two different corpora (*Penn Corpus* and *Switchboard*). Furthermore, empirical results showed that 4-5 steps of *BPTT* training was sufficient.

Hence, the techniques employed in the extended work showed that *RNNLMs* can be efficiently used in state of the art systems and that the additional processing cost by using *RNN* models does not need to be impractically high by exploiting these techniques.

Next works in the language models domain, focused on trying to improve even more the performance of the different models, having always a trade-off between speed and accuracy. In particular, when the goal is just to learn good representation of words and embeddings are the final goal, then some restrictions and architecture designs that were originally thought for language models can be slightly modified (remember that embeddings are just the weights of neurons in the network after the training is completed and they were not the final goal of language models, it was the prediction of upcoming words).

In this research line, the nonlinear hidden layer and the softmax output layer have been on the eye of researchers as the bottlenecks for improving performance. For that reason, researchers came back to the study of the original *FFNLM* and started to experiment with variations of this simple model (for example, by adding future words in the timeline as context words, by removing the non-linear hidden layer and using just a projection

layer, or by removing the restriction of keeping a probability distribution summing to 1 in the output layer, among others).

In the next sections, we will focus specifically on these word embedding models, that are intrinsically related to the language models that have been presented so far, but changing the focus of the final goal. While neural network based language models were focused on learning a predictive model able to predict upcoming words, the next models that we will see use similar neural network architectures but with some modifications specifically thought for learning word embeddings.

3.2.2.3. Word2Vec

Word2Vec, is not a method, algorithm or technique by itself. *Word2Vec* is an open-source tool[34] that takes a text corpus as input and its output is a set of features vectors (for the words contained in that corpus) with the characteristic that similar and semantically related words are projected nearby in the high dimensional vector space as it is shown in *Figure 9*.
Word2Vec provides an efficient implementation of the *Skip-Gram* and the *Continuous Bag-of-Words (CBOW)*, two popular neural network architectures originally proposed by *Mikolov et al.* in [56] for learning high-quality distributed vector representations of words that capture a large number of precise syntactic and semantic word relationships from very large data sets [56][61].
The *Word2Vec* tool also include the extensions proposed later in [61], in order to improve both the quality of the vectors and the training speed by using subsampling of the frequent words and the usage of a simple alternative to the hierarchical softmax called negative sampling.

The quality of the relationships learned by the *Word2Vec* models are measured using a new comprehensive test set specifically designed for this evaluation[35] and it is found that these models perform significantly better than *LSA* for preserving linear regularities among words. This is a very important characteristic that allows algebraic vector operations for

[34] https://code.google.com/archive/p/word2vec/
[35]http://www.fit.vutbr.cz/~imikolov/rnnlm/word-test.v1.txt

analogy reasoning like the famous example "*man is to king as woman is to queen*", which in other words means that the computation of the queen's vector can be expressed as:

$$vec(\text{“}queen\text{”}) \approx vec(\text{“}king\text{”}) - vec(\text{“}man\text{”}) + vec(\text{“}woman\text{”}).$$

Also syntactic relationships can be computed in the same way, for example:

$$vec(\text{“}smallest\text{”}) \approx vector(\text{”}biggest\text{”}) - vector(\text{”}big\text{”}) + vector(\text{”}small\text{”}).$$

And similarly, it was found that simple vector addition can often produce meaningful results, for example:

$$vec(\text{“}Russia\text{”}) + vec(\text{“}river\text{”}) \approx vec(\text{“}Volga\ River\text{”}).$$
$$vec(\text{“}Germany\text{”}) + vec(\text{“}capital\text{”}) \approx vec(\text{“}Berlin\text{”}).$$

These results suggest a complex language understanding through the computation of basic mathematical operations on the word vector representations [61].

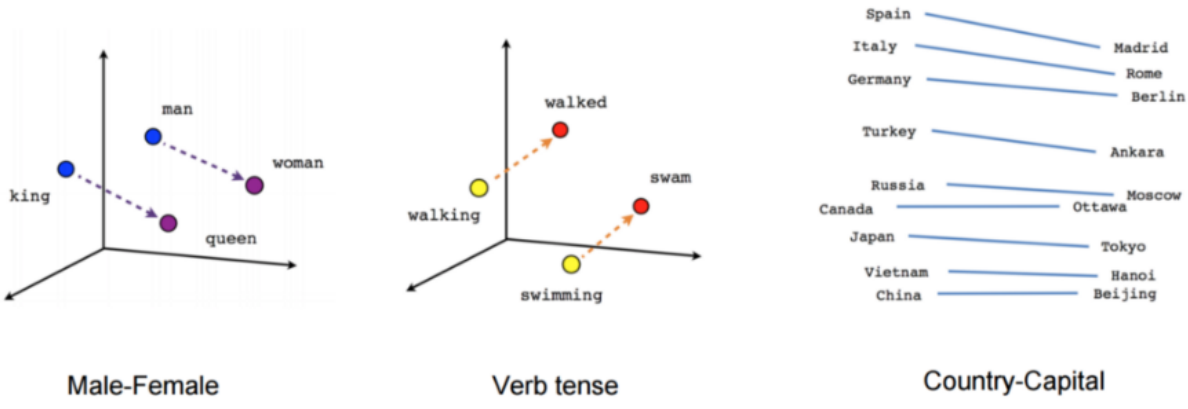


Figure 9 - Examples of syntactic and semantic linear relationships (Image Credits Tensorflow[36])

[36] https://www.tensorflow.org/tutorials/representation/word2vec1

Type of relationship	Word Pair 1		Word Pair 2	
Common capital city	Athens	Greece	Oslo	Norway
All capital cities	Astana	Kazakhstan	Harare	Zimbabwe
Currency	Angola	kwanza	Iran	rial
City-in-state	Chicago	Illinois	Stockton	California
Man-Woman	brother	sister	grandson	granddaughter

Figure 10 - Examples of some semantic relationships in the evaluation set. Source [56].

Type of relationship	Word Pair 1		Word Pair 2	
Adjective to adverb	apparent	apparently	rapid	rapidly
Opposite	possibly	impossibly	ethical	unethical
Comparative	great	greater	tough	tougher
Superlative	easy	easiest	lucky	luckiest
Present Participle	think	thinking	read	reading
Nationality adjective	Switzerland	Swiss	Cambodia	Cambodian
Past tense	walking	walked	swimming	swam
Plural nouns	mouse	mice	dollar	dollars
Plural verbs	work	works	speak	speaks

Figure 11 - Examples of some syntactic relationships in the evaluation set. Source [56].

The *Word2Vec* algorithms are sometimes called deep learning methods, and although many numerical computation and deep learning frameworks as *TensorFlow*[37] or *Deeplearning4j*[38] (among others) provide native implementations for these algorithms, formally talking, the *Skip-Gram* and *CBOW* architectures (see *Figure 12*) implement a shallow (two layers) neural network and we can not say that they are deep networks.

These models are similar but *CBOW* is used to predict target words from source context words, while the *Skip-Gram* model does the inverse and predicts source context words from the target words. Both have shown to be useful for extracting similarity of words in a text, but this inversion has the effect that *CBOW* smoothes over a lot of the distributional information (by treating an entire context as one observation). On the other

[37] https://www.tensorflow.org/tutorials/word2vec
[38] https://deeplearning4j.org/word2vec.html

hand, *Skip-Gram* treats each pair (context-target) as a different observation, and this tends to do better for larger datasets [36].
Although these models are very simple model architectures, compared to the popular neural network models (both feedforward and recurrent) the work presented in [56] concludes that it is possible to use them to train high quality word vectors.

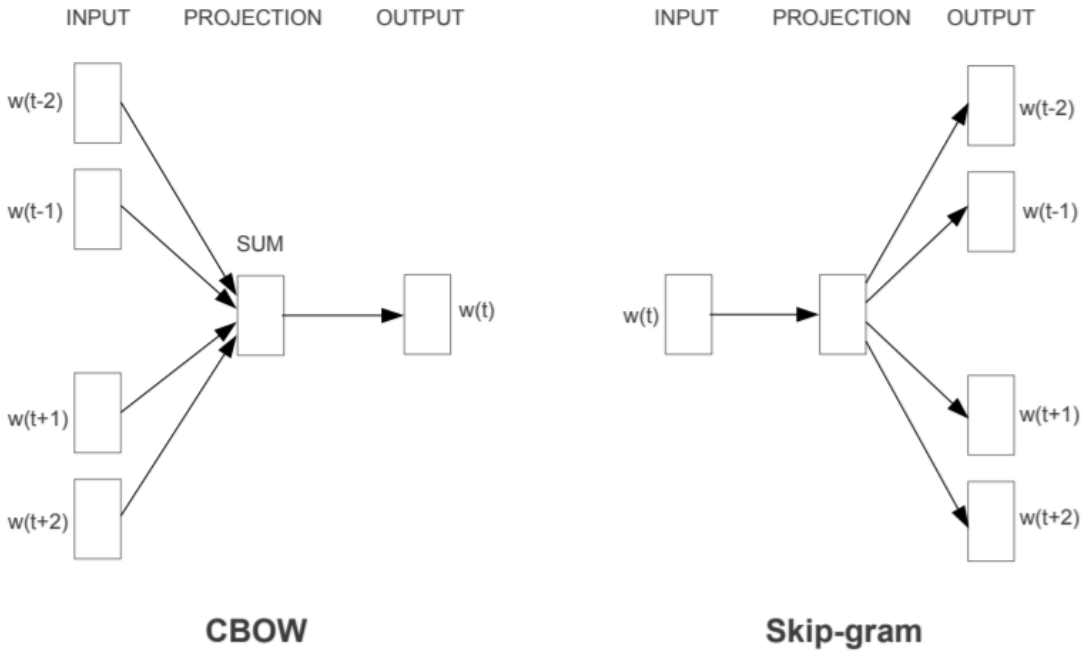


Figure 12 - Word2Vec's architectures. Source [56].

In their research, *Mikolov et al.* noted that most of the computational complexity in the architectures used in previous related works were caused by the usage of a non-linear hidden layer in the model.
Hence, they decided to explore simpler models (by removing the non-linear hidden layer) that might not be able to represent the data as precisely as deep neural networks, but can possibly be trained on much more data efficiently [56] and providing additional speedup 1000x [62]. This observation motivated the design and development of the two new model architectures, the *Skip-Gram* and the *CBOW* models, which fall in a new category called *Log-linear* models. The main goal of these new architectures is try to minimize the computational complexity while at the

same time try to maximize accuracy of the vector operations in the syntactic and semantic tasks on the evaluation set[39].

3.2.2.3.1. The Skip-Gram model

This model is inspired by the *Neural Network Language Models (NNLM)* that we described before. Like the *NNLM* models, *Skip-Gram* model tries to predict words that could be nearby (in the context) a given current word (the input word, a.k.a *target* or *middle* word). This prediction process can be used to learn embeddings for each target word (the intuition is that words with similar meanings often occur near each other in texts, and so embeddings that are good at predicting neighboring words are also good at representing similarity) [17].

In order to achieve this goal, a shallow (two layers) neural network with the characteristics shown in *Fig 8* is trained by feeding it word pairs to maximize classification of a word based on another word in the same sentence [56]. The neural network therefore learns an embedding by starting with a random vector and then iteratively shifting a word's embedding to be more like the embeddings of neighboring words, and less like the embeddings of words that don't occur nearby [17].

To illustrate this idea better, consider the example shown in *Figure 13* for a training text *"The quick brown fox jumps over the lazy dog"*. Then, if we use two words as the window size for the context, the training examples are taken from the set that results of considering couple of words (current word, context word) by moving a sliding window from the left to the right over the corpus as it is shown in *Fig 7* (input word appears highlighted in blue).

In the original paper [56], it's explained that the number of samples to feed in into the neural network for each input/target word is a random number *R* that is randomly chosen between 1 and the window size *S*. Then, *R* words from history and *R* words from the future of the current word are used as correct labels for the classification task (with the current word as input, and each of the *R* + *R* words as output). In other words, if the window size is *S* then we could potentially have *2S* training examples

[39] http://www.fit.vutbr.cz/~imikolov/rnnlm/word-test.v1.txt

(one per each combination of target and context word), but if we have *R* < *S* then some of all these possible combinations will be discarded from the training set.

In [56] also is observed that since the more distant words are usually less related to the current word than those close to it, less weight to the distant words is given by sampling less from those words in the training examples. In the previous example, if *R = 1* then for the third target word *"brown"* only one of the combinations at the left side *{(brown, the), (brown, quick)}* needs to be chosen as a new training example for the network. And the same procedure needs to be applied with the combinations at the right side *{(brown, fox), (brown, jumps)}* for choosing only one example.

It's worth mentioning that depending on the implementation used, it could be some minor details about how to select *R*. For example, in the *TensorFlow's* implementation[40] [41] not only the window size *S* can be chosen in advance, also the value of *R* (actually not directly *R*, but the *num_skips* parameter that is equals to 2*R*).

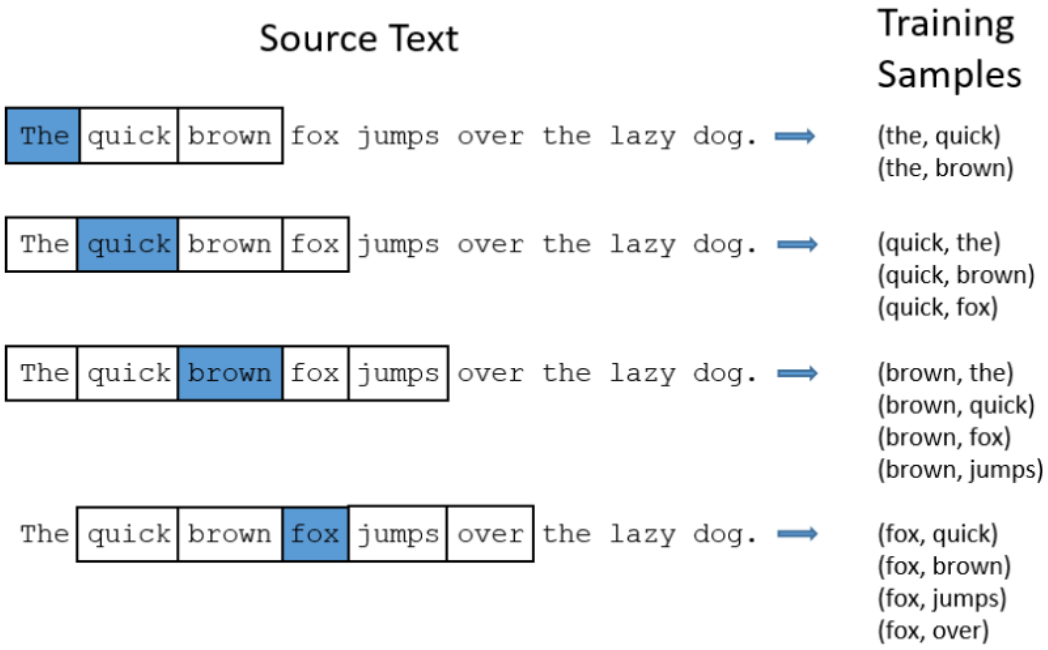


Figure 13 - Training examples to feed in into the Skip-Gram neural network model. Source [63].

[40] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/word2vec/word2vec_basic.py

[41] https://github.com/wangz10/tensorflow-playground/blob/master/word2vec.py

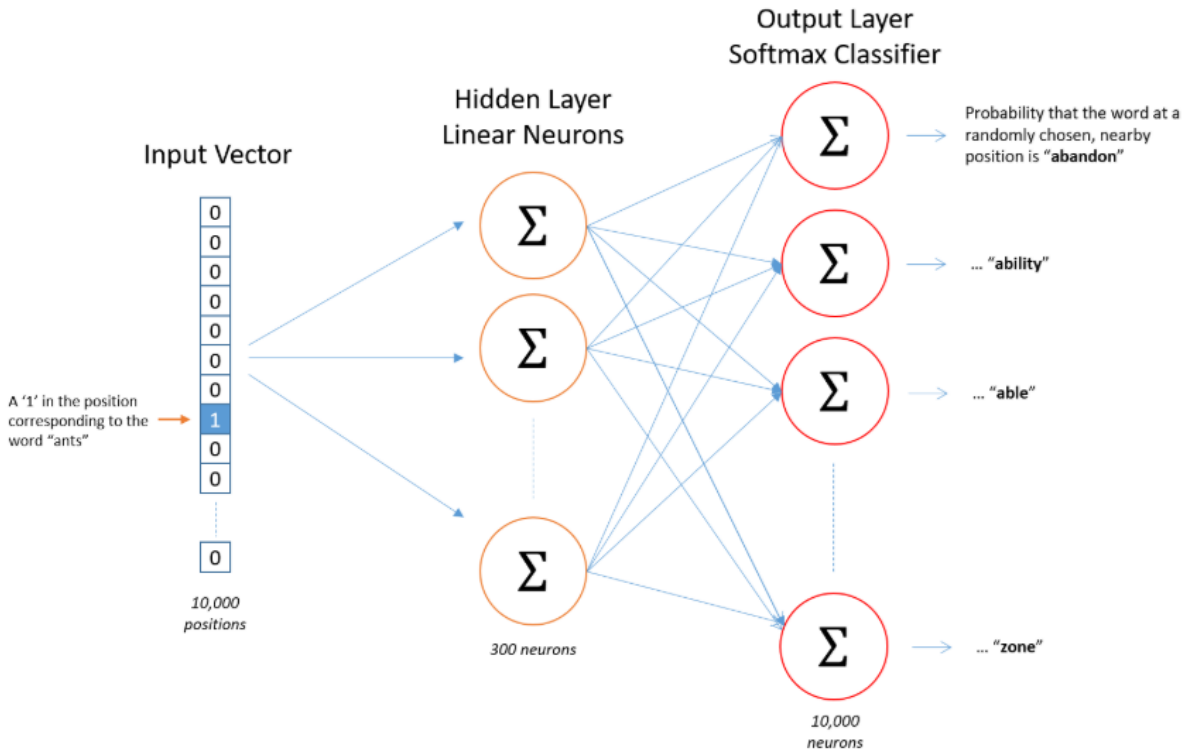


Figure 14 - Example of shallow neural network for the Skip-Gram model architecture. Source: [63].

As we can see in *Figure 14*, in the *Skip-Gram* model, the input is a single word that is represented using a one-hot encoding vector of size |*V*| (number of words in the vocabulary) where all the dimensions are 0 except the dimension that corresponds to the input word that is a 1.

Similar to a *Feed Forward Neural Network*, the input layer is fully connected to a hidden layer (all the neurons in the input layer are connected to all the neurons in the hidden layer) but without applying any activation function (that is the reason why the hidden layer in this model is also called just "projection layer").

For the output layer, one neuron per word in the vocabulary is used with a *Softmax* activation function to output the normalized probability of that word to be near the input word (in the context, before or after the input word). Hence, the output vector will have size |V| and it will actually be a probability distribution where the sum of all these output values will be 1.

About the context, it is easy to see that the window size that represent the length of the context, is an important hyperparameter to be tuned through an hyperparameter optimization phase. The research presented in the original paper [56] highlights that increasing the range (window size) improves quality of the resulting word vectors, but it also increases the computational complexity.

As we can se in *Figure 15*, there are two matrices with weights to be updated during the training process. Being *d* the hyperparameter that represents the size (dimension) of the word vectors to be learned and |*V*| the size of the vocabulary *V,* one matrix *W* of size |*V*| *x d* models the weights of the neural network for the association between the input layer with the hidden layer, and a second matrix *C* of size *d x* |*V*| models the weights of the neural network between the hidden layer with the output layer. Then, as part of the training process the weights of these two matrices *W* and *C* are updated through an efficient optimization phase while minimizing the loss function using *Stochastic Gradient Descent (SGD)* with a *Backpropagation* approach.
At the end, for each word w_j in the vocabulary (*1* <= *j* <= |*V*|), row *j* in the matrix *W* corresponds to the word embedding learned for $w_{j.}$. The reader should notice that we can simply multiply the one-hot encoded vector (of size 1 x |V|) used as input for word w_j by the *W* matrix (of size |*V*| *x d*) and the result will be the embedding vector for w_j represented by the row *j* (of size *1 x d*) of *W*. It's easy to see that by doing this, *W* will act as a selector for the embedding of word w_j and so, *W* can be seen just a lookup table for the word embeddings.

In regards to the output layer, If we consider in details the work assigned to each output neuron in this model, we realize that each of them receives one dimension value of the embedding associated to the input word from each hidden neuron, and then, all this values are multiplied by the weight associated to the two neuron's connection and aggregated to generate a temporal result that is finally passed to the softmax function. In other words, before applying the softmax activation function, each output neuron *k* is computing the dot product $v_j \cdot c_k$ between a vector v_j (embedding for w_j, output from the projection/hidden layer corresponding to row *j* in *W*) and a c_k column in *C* (context embeddings that are helpful for the fake task of classifying and predicting neighbors words as part of

the training processes while learning the really important goal that are the weights of W, the word embeddings).

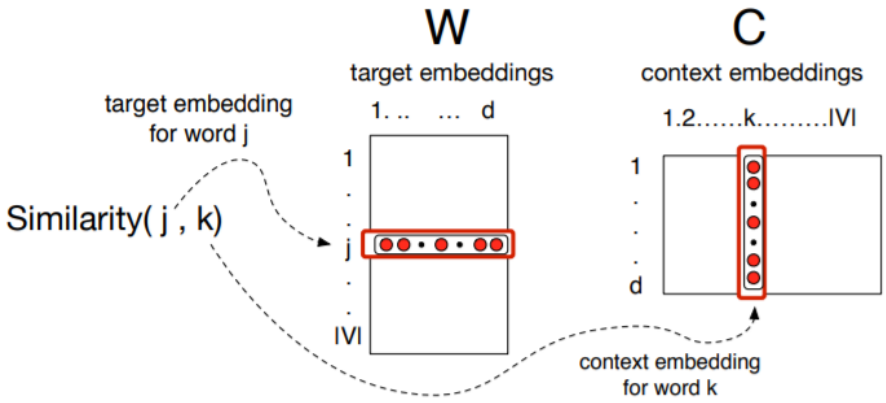


Figure 15 - W and C matrices with the learned embeddings for the target and context words respectively. Source [17].

It is well known that the higher the dot product between two vectors, the more similar they are. Then, based on this simple observation and leveraging the maths that is modeled by the smart architecture behind the *Skip-Gram* neural network model, its objective is to maximize the probability of finding w_k in the context of w_j. And since the raw dot product between the two vectors is not a probability, a softmax function is applied. Also, since we want a probability distribution where the sum of all the output neurons is 1, a division by the sum of the results from all the others $|V|$ output neurons is applied.

$$p(w_k|w_j) = \frac{exp(c_k \cdot v_j)}{\sum_{i \in |V|} exp(c_i \cdot v_j)} \cdot \quad \text{(Eq. 15)}$$

It is easy to see that in order to maximize *Equation 15*, the dot product in the numerator needs to be as higher as possible and on the contrary a small denominator is better. Hence, in each training iteration, *Skip-Gram* will adjust the network weights for *W* and *C* matrices in order to maximize this function as much as possible (making word's vector close to the words that occur near it, that is, in the numerator, and further from every other word, that is, in the denominator [17]). Once the training phase

is completed, matrix *C* can be discarded and the learned weights in *W* can be used as good embeddings for words in vocabulary *V*.

3.2.2.3.2. The CBOW model

The *CBOW* model is very similar to the previously presented *Skip-Gram* model, except that *CBOW* predicts target words from source context words, while the *Skip-Gram* does the inverse and predicts source context-words from the target words [36]. As an illustrative example, considering the example shown in *Fig 7,* using a windows size of length 2, *CBOW* predicts target/middle word *"fox"* from source context words *"quick brown jumps over"*.

The original *CBOW* architecture shown in *Figure 16* is similar to the standard bag of words, where order doesn't matter. It tries to predict the middle word by averaging the embeddings of the context's words through a projection layer as it is shown in *Figure 17*. Since the context changes continuously by moving a fixed length window through the corpus from left to right, then the *"C"* in *CBOW* comes from *"Continuous"* distributed representation of the context.

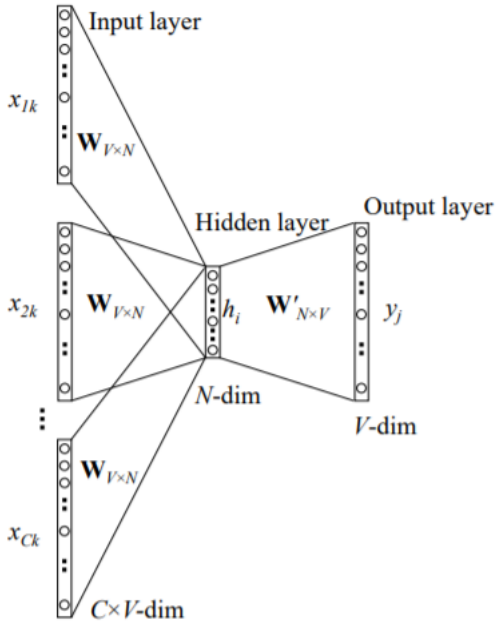


Figure 16 - CBOW architecture. Source [64].

Recently, in [65], one of the tricks presented by the *Facebook AI Research* in order to train high-quality word vector representations consists in the usage of a *position-dependent weighting* schema. By doing this, the behaviour of the standard *CBOW* model is slightly modified, moving from a non weighted approach (order does not matter for words in the context) to a weighted one (the contribution of each word to the average is weighted depending on its relative position inside the context). Later in this work, as part of the main research presented in this thesis, a similar weighting schema based on [66] is applied to the *CBOW* model in order to improve its performance.

Other difference between *CBOW* and the *Skip-Gram* model is the number of training samples to feed into the neural network each time the context windows moves from the left to the right over the corpus. While *Skip-Gram* generates a training sample for each word in the context (as it is shown in *Fig 7*), *CBOW* generates only one sample for the whole context since all the context's words are feeded into the neural network at the same time. This difference has the effect that *CBOW* smoothes over a lot of the distributional information (by treating an entire context as one observation). On the other hand, since *Skip-Gram* treats each pair (context-target) as a different observation, this tends to do better for larger datasets [36].

Similar to the *Skip-Gram* model, *CBOW* uses an auxiliary classification phase through a shallow (one hidden/projection layer) neural network to update iteratively the neuron's weights in the hidden layer. The training criterion is to correctly classify the current (middle) word [56]. When *CBOW* computes the output for the hidden layer, instead of directly copying the input vector of the input context word, *CBOW* takes the average of the vectors of the input context words, and it uses the product between this averaged vector and the weight matrix between the input and the hidden layer as the output for the hidden layer [64].

It's worth mentioning that despite in the original paper the average of the context words is always used, there are some implementations like the one provided by the *Gensim* package for *Python* (used later in this

work) that allows to configure whether the average is used or just the sum of the vectors of the context words (boolean 'cbow_mean' parameter)[42].

Finally, the output layer is computed using the same approach that was already explained for the *Skip-Gram* model, that is, one output neuron per word in the vocabulary, which in this case (for the *CBOW* model), it outputs the probability (using a softmax classifier) of that word to be the target/middle word for the input context.

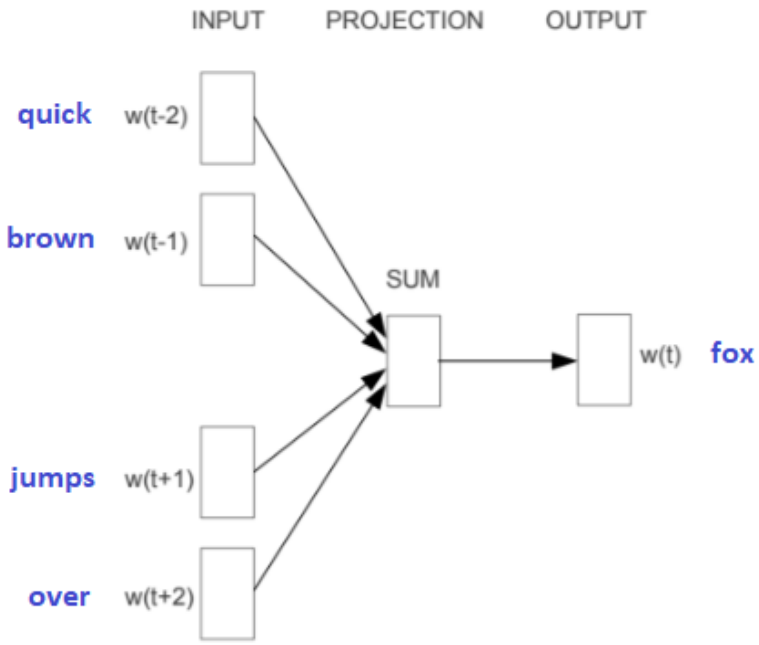


Figure 17 - CBOW model [56] predicts "fox" from "quick brown jumps over" [63].

At the end, after the training phase is completed, the rows in the updated weight matrix between the input and the hidden layer is used as the learned word vectors.

3.2.2.3.3. Optimizations to the original models

The first version of the *Skip-Gram* and *CBOW* models were originally presented in [56] with the characteristics previously described

[42] https://radimrehurek.com/gensim/models/word2vec.html

here. Later, in [61] authors published a second paper that extends the first one with several extensions that improve both the quality of the vectors and the training speed [61].

The main improvements are related to:

I. adding of common phrases that are repeated in the corpus as single tokens to the vocabulary

II. subsampling of frequent words

III. usage of a different loss function called *"Negative Sampling"*

Common phrases as single tokens

Authors suggested to consider frequent phrases in the corpus as single tokens and add them to the vocabulary as single new words. This suggestion comes up after observing that there are a lot of cases where a the semantic of a phrase is not directly related to the semantic of its individual words. Some examples are: *"New York Times"* (newspaper), *"Golden State Warriors"* (NBA team), *"Australian Airlines"* (airline), *"Steve Ballmer"* (company executive).

The method employed for phrase detection works by considering bigrams *(w_i, w_j)* that are consecutive pairs of words. The idea is to count the number of occurrences of the bigram and compare it with the number of occurrences that it should have if w_i and w_j were not related (their occurrences should be independent events). Hence, as shown in *Equation 16*, a formula based on the unigram and bigram count is used for phrase detection.

$$\text{score}(w_i, w_j) = \frac{\text{count}(w_i w_j) - \delta}{\text{count}(w_i) \times \text{count}(w_j)}. \qquad \text{(Eq. 16)}$$

In *Equation 16*, δ is used as a discounting coefficient and prevents too many phrases consisting of very infrequent words to be formed.

Finally, the bigrams with score above a chosen threshold are then used as phrases by adding them to the existing vocabulary as a single word. The process is repeated (generally between 2 and 4 times) over the training data using a decreasing threshold value approach. By doing this, in each new iteration new bigrams appear, allowing longer phrases that consists of several words to be formed [61].

Subsampling of frequent words

The idea behind this optimization is to remove words that can be considered as *"noise"* because they do not add any relevant information. Common words like *"the"*, *"and", "is", "this"* (among others) are examples of this kind of frequent words. And this is because *Word2Vec* models can benefit much better when training the neural network using words that are in the same context and share some meaning (like *"France"* and *"Paris"*) than when using words that are in the same context but not share some semantic relationship (like *"France"* and *"the"*).

Also, since frequent words appear in most of the contexts, then their vector representations are too generic and it was observed that they do not change significantly after training on several million examples [61]. In other words, the update process that occurs during the training phase can not move the vectors associated to frequent words to any specific direction because they are nearby to many different kind of words and they do not belong to any specific domain or field. Then, those vector representations for frequent words are not relevant for finding similarities or analogical reasoning.
Because of these issues about frequent words mentioned before, authors in [61] presented results showing that by removing frequent words the *Word2Vec* models they were able to improve both, the quality of the learned vectors and also the training speed.

Naturally, the subsampling technique used is related to the frequency of occurrences of words in the corpus. *Equation 17* shows the formula that is formally used to compute the probability of discarding a word during the subsampling process.

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}} \,. \qquad \text{(Eq. 17)}$$

In *Equation 17*, $f(w_i)$ is the frequency of word w_i and t is a chosen threshold (typically around 10^{-5}). Authors argue to have chosen this formula because it aggressively subsamples words whose frequency is greater than t while preserving the ranking of the frequencies.

Negative sampling

If we observe the *Softmax* objective function that the *Word2Vec* models try to maximize in *Equation 15*, we realize that there is a high computational cost in computing this function for each training example, mainly because of its denominator that implies to compute a dot product for every word in the vocabulary *V* (the computational cost increases linearly with the numbers of words in the vocabulary). This is impractical, particularly when using very large datasets.

Many researchers have studied different approaches for replacing the full *Softmax* objective function. In [67], *Morin* and *Bengio* proposed an approximation to the *Softmax* function called *Hierarchical Softmax*. *Hinge loss* was used by *Collobert* and *Weston* in [68]. *Noise Contrastive Estimation (NCE)* was suggested by *Mnih et al*. in [69], and based on this last one, the *Negative Sampling* technique was proposed in [61] as an optimization for the original *Word2Vec* models.

The *Negative Sampling* proposal consists on using an alternative objective function to improve the cost of computing the original objective function for each training example. The idea behind this approach is to avoid the update of all words in the vocabulary, and instead, just pick a small number *K* of *negative samples* to update. Values of *k* in the range 5–20 are useful for small training datasets, while for large datasets the *k* can be as small as 2–5 [61].
The concept of *negative* comes from the fact that *K* samples to update are taken from words which their associated neurons in the output layer are expected to be 0. One additional update is reserved for updating the

positive word, that is, the word which its neuron in the output layer is expected to be 1.

When using the *Negative Sampling* approach, the selection of the *K* words is based on a weighted unigram distribution raised to the 3/4rd power, in other words, the probability for selecting a word w_i as a negative sample is related to its frequency $f(w_i)$, with more frequent words being more likely to be selected as negative samples.

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=0}^{n} \left(f(w_j)^{3/4} \right)} \quad . \qquad \text{(Eq. 18)}$$

According to the authors in [61] the value of 3/4rd power used in *Equation 18* was the best option obtained empirically after executing different experiments and outperformed significantly the unigram and the uniform distributions. In the previous equation, the denominator is added just to keep a probability value between 0 (w_i is not present in the corpus) and 1 (w_i is the only word present in a size 1 vocabulary).

Although *Negative Sampling* is based on *NCE*, and *NCE* approximately maximizes the log probability of the softmax, this property is not important for the *Word2Vec* models because the main goal for them is to learn high-quality vector representations. For that reason authors in [61] simplified *NCE* in such way that the simplification does not give guarantees of maximize the log probability of the softmax (the resulting dot products will not produce optimal predictions of upcoming words [17]) but the vector representations retain the quality, and that is the most relevant contribution. That said, the *Negative Sampling* objective function is defined as:

$$\log \sigma({v'_{w_O}}^{\top} v_{w_I}) + \sum_{i=1}^{\kappa} \mathbb{E}_{w_i \sim P_n(w)} \left[\log \sigma(-{v'_{w_i}}^{\top} v_{w_I}) \right], \qquad \text{(Eq. 19)}$$

where the v'_{wo} corresponds to the word embedding of the context word taken from matrix *C* in the output layer, v_{wI} to the word embedding of the

input word taken from matrix *W* in the hidden layer, and v'_{wi} to the word embedding of the negative sample *i* (1 <= *i* <= *k*) taken from matrix *C* with probability $P_n(w_i)$. σ represents the sigmoid function defined in *Equation 12* that is used by the logistic regression to differentiate data from noise.

For the *CBOW* model, since its architecture can be seen as a mirror or inverted version of the *Skip-Gram* architecture, then, we can think v_{wI} as the sum of the word embeddings of the inputs (words in the context) and v'_{wo} as the word embedding for the target (middle word).

Finally, in order to maximize the *Negative Sampling* formula we need to maximize the dot product of the word with the actual context words, and minimize the dot products of the word with the *k* negative sampled. By doing this, after the model is trained, the embeddings of words that generally share similar contexts tend to be similar (the input words needs to have similar weights to allow the neural network to output similar contexts in the Skip-Gram model), and on the other hand, embeddings of words that rarely shared same context tend to be very different (if not, the network would output similar contexts).

3.2.2.4. App2Vec

So far, different techniques for word embedding have been presented in this work, and all of them have their basis in the *NLP* field, strongly motivated by the distributional hypothesis, suggesting that words that are used and occur in the same contexts tend to purport similar meanings [33], a concept popularized by *John R. Firth* in [35] with the phrase *"a word is characterized by the company it keeps".*

App2Vec[66] differ from the previous techniques in its conception, the application domain and the goal to solve. While the goal of the previous techniques presented in this work were to find good vector representations (embeddings) of words by studying how they are used in different corpus and contexts, *App2Vec* has as main goal to model mobile applications as vectors based on how users use those apps.
And although, it seems a totally different domain (mobile applications vs *NLP*) *App2Vec*'s authors realized that many ideas from the *NLP* field could

be taken in order to model mobile applications as vectors. This characteristic makes *App2Vec* to be very attractive in the scope of this work, since a similar scenario, using word embedding techniques out of *NLP* domain is used later when trying to find the best vector representation of Internet Domain Names, one of the main goals of this thesis.

Hence, in order to map the *NLP* problem of finding the vector representation of words using word embeddings techniques (like the ones implemented by *Word2Vec*), *App2Vec* applies the following mapping between concepts:

- The concept of *word* is mapped to a *mobile app session*
- The concept of *document* is mapped to *the sequence (ordered) of all mobile app sessions for an specific user*

Furthermore, when tracking user activities (like the use of mobile applications) there are special properties in the sequences of activities that users performs. Two of the most important are:

- The elapsed time gap between consecutive activities can vary and it is very important. Two activities executed very close in time probably are much more related than two consecutives activities that were not executed immediately one after the other
- The same activity can be executed more than once consecutively. For example, considering the apps *{a, b, c}*, then the sequence *(a, a, a, b, b, c, a)* is totally valid.

From the observations above, before executing the *App2Vec* vectorization algorithm, a preprocessing phase is needed for a) incorporating an elapsed time gap variable and b) the removal of duplicates for same consecutive actions. If we consider the same example presented above, after adding the time gaps and removing duplicates, the final sequence is *(a, g_1, b, g_2, c, g_3, a)*, where *a, b, c* represent mobile apps, and g_i represents the elapsed time gap between two consecutive mobile app sessions. In the *Python* implementation of *App2Vec* (kindly provided by its authors), the new text format needed as input is explained through the following example:

bird g:2 elephant g:10 car g:100 snake g:11 chicken g:12 snail g:13 sheep g:14

More formally, what *App2Vec* wants to solve is to learn a similarity function *sim(ai, aj)* for two apps *ai* and *aj*, given a set of users *U*, and the historic app usage sessions S_u of a user $u \in U$. Each app session $s_i \in S_u$ is represented by (u_i, a_j, t_s, t_e), meaning user u_i used app a_j starting at time t_s and ending at time $t_{e\ [66]}$.

In order to achieve its goal, *App2Vec* uses a slight modification of the original *CBOW* model (already presented in Section 3.2.2.3.2*)*, where the main change consists in using the time gaps between app sessions to create a weighted schema inside the context window, before averaging them in the projection layer. The formula chosen by *App2Vec* for weighting a word w_i to the target word w_t is defined as:

$$r(w_i, w_t) = \alpha^l, \qquad \text{(Eq. 20)}$$

where α is empirically chosen as 0.8, and l is the amount of the gap (*e.g.*, number of minutes) between app w_i within the current context and target app w_t.

In other words, what *App2Vec is* basically doing is changing the standard formula used in the *CBOW* model for averaging input vectors from this:

$$\bar{v} = \frac{1}{2c} \sum_{-c \le j \le c, j \neq 0} v_j \ , \qquad \text{(Eq. 21)}$$

(where c represents the context window size and v_j the context word vectors) to this:

$$\bar{v} = \frac{\sum_{-c \le j \le c, j \neq 0} r(w_j, w_t) v_j}{\sum_{-c \le j \le c, j \neq 0} r(w_j, w_t)} \ , \qquad \text{(Eq. 22)}$$

(where *c* represents the context window size, v_j the context word vector and $r(w_j, w_t)$ the weighted value described above to consider the time gap between w_j and w_t).

After the modified vector is fed into the projection layer, no other changes to the original *CBOW* model take place. According to the results presented in the *App2Vec*'s paper [66], the weighted schema have demonstrated to outperform the standard bag-of-words approach, achieving an improvement of 37% on the task of capturing semantic relationships between apps.

Recently, in [65], *Facebook AI Research* suggested the usage of weighted schemas as one possible trick for improving the quality of word vector representations, validating in this way the approach followed by *App2Vec*.

3.2.2.5. FastText

Like *Word2Vec*, *FastText* is not a method, algorithm or technique by itself. *FastText* is an open-source, free, lightweight library that allows users to learn text representations and text classifiers [70]. This tool comes in two flavors, it can be used either in unsupervised mode to learn word embeddings or in a supervised mode using a labeled dataset (one label and sentence per line) to train a text classifier.
Developed at *Facebook Research*, *FastText* is an extension of the work previously done with *Word2vec* by *Mikolov et. al. at Google Research*. Its fundamentals are explained in the papers [71], [72] and [73]. Like *Word2Vec*, the unsupervised mode for learning word embeddings in *FastText* can be executed with both models: *Skip-Gram* or *CBOW*.

The main principle behind fastText is that the morphological structure of a word carries important information about the meaning of the word, which is not taken into account by traditional word embeddings, which train a unique word embedding for every individual word [74]. And that is the main difference between *FastText* and *Word2Vec,* that is, *FastText* can learn from subwords.

In order to do that, *FastText* adds the notion of *bag of characters* (a.k.a known as *n-grams*), which are defined by using two of the most important hyper-parameter that *FastText* has, *minn* and *maxn*. These parameters can be configured when running the unsupervised mode of *FastText*, and they are related to the range of size for the subwords. In other words, subwords are all the substrings contained in a word between the minimum size (*minn*) and the maximum size (*maxn*). Also, the word itself is considered to be in the set of its n-grams [71]. If *maxn* is set to 0, or lesser than *minn*, no character *n-grams* are used, and the model effectively reduces to *Word2Vec* [74].

In order to understand better the concept of *n-grams*, consider the word *"where"* and *minn* = *maxn* = *3*. Then, this word will be represented by the character *n-grams*: <*wh, whe, her, ere, re*> (note the special boundary symbols < and > at the beginning and end of words), and the special sequence <*where*>.

Finally, the learning phase for updating a vector representation associated to a word is computed by taking into account the word's morphology, which is modeled aggregating all the vector representations of its *n-grams* (and the word itself) [71]. This, can be expressed more formally using *Equation 23* to represent the vector representation of word *w*.

$$v_w + \frac{1}{|N|} \sum_{n \in N} x_n .$$

(*Eq. 23*)

In *Equation 23*, v_w is the current vector representation of *w*, *N* is the set of n-grams taken from *w*, and x_n is the vector representation for the specific n-gram *n*.

The main advantages of *FastText* compared to *Word2Vec* are:

- **Better performance at syntactic tasks.** This is due *FastText* considers a word as the aggregation of different subwords, whereas *Word2Vec* considers each words as a single and indivisible token.

- **Support for o*ut-of-vocabulary* (*OOV*) words**. The *OOV* scenario is not supported when using *Word2Vec*, but with *FastText* the only requirement is that exist at least one match between any subword (of the *OOV* word) and any subword used during the training phase (by any word in the vocabulary) . As an example, consider the word *"cartoon"* which does not belong to the vocabulary, but *"career"* does. Then, (supposing *minn=3*) a vector representation for *"cartoon"* can be computed since there is at least one match for the substring *"car"* that is present in both *"career"* and *"cartoon"*.

- **Good representation for rare words**. Despite a word can have very few occurrences, probably its subwords can be present in many other words of the vocabulary (as substring), then the final aggregation of the subwords for the rare word would probably end up getting a good vector representation, much better than the one given by *Word2Vec*.

As disadvantage (compared to *Word2Vec*) it can be said that the model's complexity is increased by adding two new important parameters that needs to be tuned very well in order to get good results. The *minn* and *maxn* parameters that control the *n-grams* length can vary depending on the domain.
For instance, for english language the default values of *minn=3* and *maxn=6* achieve good results in practice when training using large corpuses [75], but in other scenarios like the research about Internet domain names embeddings that is presented in Chapter 4 those values are really bad (the performance is worse than *Word2Vec*) and exhaustive experimentations needed to be done in order to find good settings for them (*minn = 11, maxn = 17*). This example evidences the fact that the hyper-parameter tuning phase for finding a good model is more complex and time consuming when tuning a *FastText* model than a *Word2Vec* model.
And last but not least, it's worth mentioning that splitting and training with subwords increases the training time and the resource usage (ram mainly) considerably, also the generated models using *FastText* with *n-grams* have sizes considerably bigger than models generated by *Word2Vec*.

3.2.3. GloVe: Global Vectors for Word Representation

GloVe is an unsupervised learning algorithm for obtaining vector representations for words [76]. It was created at *Stanford University* by *J. Pennington, R. Socher, and C. D. Manning* in 2014 and presented in the paper "*GloVe: Global Vectors for Word Representation"* [19].

This learning algorithm can be seen as an hybrid approach and it can not be assigned exclusively to any of our previous classification of count-based or prediction-based models. Since it builds a word-word co-occurrence matrix, then we are tempt to categorize it as a count-based model, but since it uses Stochastic Gradient Descent (SGD) to minimize a cost function and it captures not only words similarities, but also the semantic analogies between words (in the same way like the neural predictive models work), we are also tempt to classify this technique as a prediction-based model approach. Authors argue that *Glove* leverages the best of the two worlds, that is: it has the advantage of capturing the global statistics of word-word co-occurrences like the count-based methods, and also the efficiency of the prediction-based models by simultaneously capturing the meaningful linear substructures prevalent in recent *log-bilinear* prediction-based methods like *word2vec* [19].
For this reason we have decided to separate this learning algorithm into a different categorization, an hybrid approach. It's worth mentioning that authors arrives to the conclusion that *Glove* outperforms other models on word analogy, word similarity, and named entity recognition tasks.

How does GloVe work?

The main approach in the design of *Glove* is the usage of global statistics of word-word co-occurrences, but considering hidden linear relationships substructures present in word vectors spaces.
Pennington et. al noted that clustering structures of similar words are not the only important substructures present in the word vector space. Nearest neighbors using a single scalar value like the euclidean or cosine distance are not good for summarizing very complex linguistic similarities (for example, that the relationship between man and woman can represent the concept of gender), mainly in a high dimensional vectors scenario. For

such high dimensional vectors there is a more natural approach to summarize relationships between words: vector difference between two words vectors [77].

As an example we can consider the vector representations of some countries and their capitals. *Figure 18* shows the 2-dimensional PCA projection of these word representations.

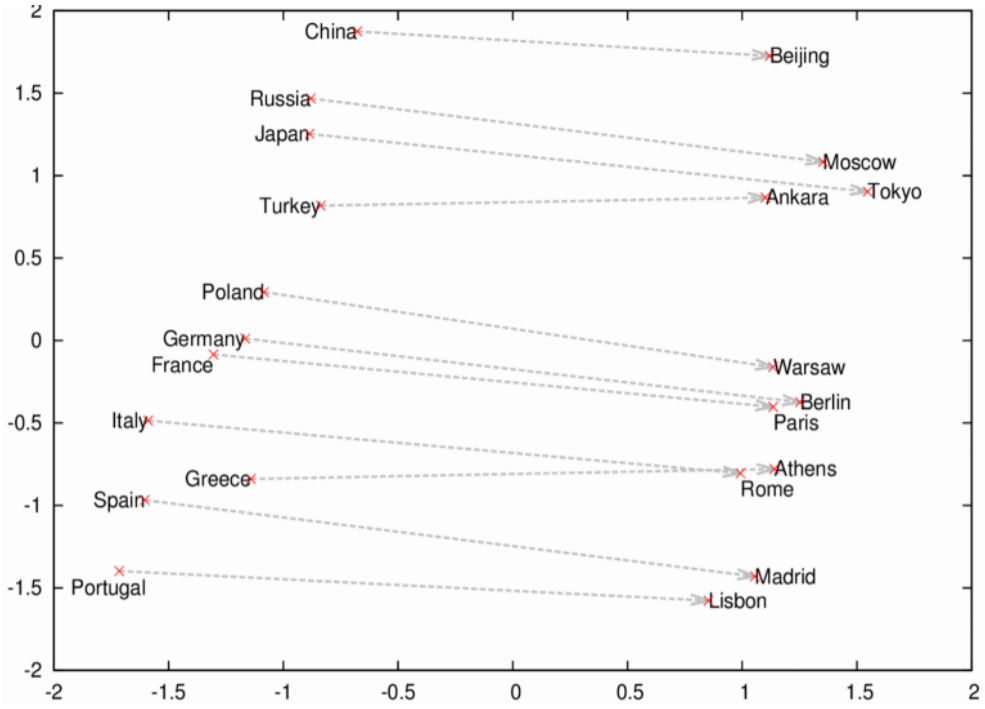


Figure 18 - 2 dimensional PCA projection of countries and their capitals. Source [78].

By seeing the image above we can realize that not only cities or countries form dense clusters, but also that in all cases the difference between the vector representation of a country and the vector representation of its capital are more or less the same, and it is capturing some abstract concept that relates a country with its capital.
These linear relationships had been previously observed in [43] and it has became one of the most popular benchmarks for word embeddings after Mikolov et al. in [79] showed that proportional analogies (*a* is to *b* as *c* is to

d) can be solved by finding the vector closest to the hypothetical vector calculated as *c* - *a* + *b* (e.g. *king* - *man* + *woman* = *queen*) [80].

Other important observation pointed out in *Glove* is that *ratios* of co-occurrences can encode important meaning. This, complements the previous observation about the linear substructures in the word vector space, and both will be important for defining the Glove's model.
But, before presenting the model, let's illustrate the importance of *ratios* through a simple example with two concepts *i* = *ice* and *j* = *steam* from the thermodynamic phase and compute how often they occur with other words (contexts) *x* = *{solid, gas, .. ., water}*

	x = solid	*x = gas*	*x = water*	*x = random*
P(*x* | *ice*)	large	small	large	small
P(*x* | *steam*)	small	large	large	small
P(*x* | *ice*) / *P*(*x* | *steam*)	**large**	**small**	~1	~1

Table 5 - Probability of word co-occurrences and its ratios (simplified view of the original table in [19])

In *Table 5*, the probability *P* that measures the probability that word *j* occurs in the same context than the word *i* is denoted by *P*(*j* | *i*) and it can be computed over a big corpus by taking the ratio between the number of times that *j* co-occurs in the context of *i* and the total number (sum) of co-occurrences of any word in the context of i [19]. If we think in the word-word co-occurrence matrix *X*, then:

$$P(j|i) = \frac{X_{ji}}{\sum_{k=1}^{|V|} X_{ki}} \quad , \qquad \text{(Eq. 24)}$$

where the numerator X_{ji} corresponds to the number of times that word *j* co-occurs in the context of word *i* and the denominator is just the sum of all the cells in the column *i* of *X* (representing the total number of co-occurrences of any word *k* in the context of *i*).

By seeing the previous table, we can note that when computing the ratio using a word *x* that is related to both ice and steam, then the result is close to 1 (because the numerator and denominator tends to be cancelled since their values will be high and very similar).
In a similar way, we can note that when *x* is a random word without any relationship with ice or steam, then the result is close to 1 too (but now because a similar low value is found at the numerator and denominator causing them to be cancelled).
By continue analyzing these ratio results, it's interesting to see what happen when *x* is related to just one of the words ice or steam. For instance, when x = solid, the ratio Px_i / Px_j is large, and when x = gas, the ratio is small. In both cases, they are very far from the cancellation value of 1.
With these observations in mind, we can realize that the ratio is better than the raw probabilities in the task of distinguish relevant words (solid and gas) from irrelevant words (water and fashion). This suggests that the appropriate starting point for word vector learning should be with ratios of co-occurrence probabilities rather than the probabilities themselves [19].

Hence, based on the previous two important observations about linear substructures in the word vector space and that ratios between co-occurrence probabilities can encode meaning, authors propose *Equation 25* as the objective function to be minimized by *Glove* during its training phase.

$$J = \sum_{i,j=1}^{|V|} f(x_{ij})(w_i^T \cdot w_j - logXij)^2 \ . \qquad \text{(Eq. 25)}$$

In *Equation 25*, |*V*| is the size of the vocabulary and $f(x_{ij})$ is a weighting function that needs to meet some requirements: $f(0)$ should be equal to 0, $f(x)$ should be non-decreasing so that rare co-occurrences are not overweighted and $f(x)$ should be relatively small for large values of *x*, so that frequent co-occurrences are not overweighted. An example of this function that worked well is:

$$f(x) = \begin{cases} (x/x_{\max})^{\alpha} & \text{if } x < x_{\max} \\ 1 & \text{otherwise} \end{cases} , \qquad \text{(Eq. 26)}$$

where α = ¾ empirically demonstrated to give the best results.

Since we previously had noted that $P(i \mid j) = X_{ij}$ then in order to minimize this objective function *J*, *Glove* tries to satisfy the log bilinear model as part of the optimization process:

$$w_i \cdot wj = log\ P(i|j)\ , \qquad \text{(Eq. 27)}$$

where w_i and w_j are the vector representations for words *i* and *j* respectively and $P(i \mid j)$ is the conditional probability of the co-occurrence of word *i* in the context of word *j*.

Since w_i is a row vector in the matrix, then the transpose of the vector is used in the objective function, and also the weighting function *f(x)* is added so the most frequent terms that do not add too much meaning have less impact in the results.
Hence, by making the dot product between words embedding equal to the log probability of their co-occurrence, the following equation is true:

$$w_x \cdot (w_a - w_b) = w_x \cdot w_a - w_x \cdot w_b\ . \qquad \text{(Eq. 28)}$$

Then, since the model was trained to satisfy *Equation 27*, we have:

$$w_x \cdot w_a - w_x \cdot w_b = log\ P(x|a) - log\ P(x|b)\ , \qquad \text{(Eq. 29)}$$

and this (*Equation 29*) can be re-written as (because of the difference property of logarithms):

$$log\ P(x|a) - log\ P(x|b) = log\ \frac{P(x|a)}{P(x|b)}\ . \qquad \text{(Eq. 30)}$$

Hence, we arrive to:

$$w_x \cdot (w_a - w_b) = log\ \frac{P(x|a)}{P(x|b)}\ . \qquad \text{(Eq. 31)}$$

By analyzing *Equation 31*, we figure out that we can take any word *x* from the vocabulary and by doing the dot product with other two word vectors *a* and *b*, the result is a log function of the ratio between the co-occurence of this word *x* with words *a* and *b*, which we previously observed that carried important information about the meaning and relationship between words.

Finally, it is important to highlight that this smart definition of the model allows to *Glove* to performs fast training, to scale to huge corpora and to get good results even with small corpus and small vectors [77].

3.2.4. Summary of document and word embeddings techniques

This section summarizes the techniques that were presented in this chapter for document and word embeddings. It is worth mentioning that in this work we will focus on the techniques regarding embeddings to the word level, therefore embeddings to the whole document level are not studied exhaustively and just some basic techniques are presented (mainly those that are related with some of the word embeddings algorithms that are studied). The reader can see *Skip-Thought Vectors [81]* or *Doc2vec [82]* for more sophisticated embeddings techniques for whole sentences or documents.

Technique	***Category***	***Year***	***Papers***	***Observations***
One-Hot encoding	Word embedding	-	-	Traditionally used for categorical data embedding
Term-Document Matrix	Word and Document embedding (count-based approach)	1972	The SMART retrieval system: experiments in automatic document processing [30]	Rows can be used as word embeddings and columns as documents embeddings (a.k.a bag of words)
Pointwise Mutual	Word and	1989	Word association norms,	Weighted version

Information (PMI)	Document embedding (count-based approach)		mutual information, and lexicography [38]	of the original Term-Document Matrix
Term frequency - Inverse document frequency (TF-IDF)	Word and Document embedding (count-based approach)	1957 1972	A Statistical Approach to Mechanized Encoding and Searching of Literary Information [83] A statistical interpretation of term specificity and its application in retrieval [84] Other important papers: The Automatic Creation of Literature Abstracts [85] Relevance weighting of search terms [86] Precision Weighting—An Effective Automatic Indexing Method [87] Probabilistic models of information retrieval based on measuring the divergence from randomness [88]	Weighted version of the original Term-Document Matrix. The combination of two factors: • Term frequency (Luhn 1957): frequency of the word (can be logged) • Inverse document frequency (IDF) (Sparck Jones 1972)
Latent Semantic Analysis (LSA)	Word and Document embedding (count-based approach)	1988 1990 1997	Computer information retrieval using latent semantic structure [89] Indexing by latent semantic analysis [47] A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge [48]	Dimensional reduction applied to the original Term-Document matrix M $M \simeq V.K.C$ Rows in the V matrix represent word embeddings and columns in the C matrix document embeddings
SVD applied to the Term-Term Matrix	Word embedding (count-based	1992 1996 2002	Dimensions of meaning [90] Producing high-dimensional semantic spaces from lexical	Similar to LSA but dimensional reduction is applied to the

	approach)		co-occurrence [42] Detecting Patterns in the LSI Term-Term Matrix [91]	Term-Term matrix. a.k.a Truncated SVD on Term-Term matrix
N-grams	Language modelling (probabilistic approach based on statistics of bigram, trigrams, etc)	1955 1948	Example of a Statistical Investigation of the Text of “Eugene Onegin” Illustrating the Dependence Between Samples in Chain [51] A Mathematical Theory of Communication [92]	Simplest language model. Language models give the basis of current word embedding techniques.
Feedforward Neural Language Model (FFNLM)	Language modelling and Word embedding (predictive approach)	2003	A neural probabilistic language model [50]	Original employed for language modeling based on deep neural networks.
Recurrent Neural Network Language Model (RNNLM)	Language modelling (predictive approach)	2010	Recurrent Neural Network Based Language Model [57]	Original employed for language modeling based on neural networks
Word2Vec	Word embedding (predictive approach)	2013	Efficient estimation of word representations in vector space [56] Distributed representations of words and phrases and their compositionality [61]	Simplification of FFNLM (shallow network without non-linear activations). Two architectures: CBOW and Skip-Gram
App2Vec	Mobile apps embedding (predictive approach)	2016	App2Vec: Vector modeling of mobile apps and applications [66]	Weighted version (based on time gaps) of the CBOW architecture of Word2Vec
FastText	Word embedding and supervised text classification (predictive approach)	2016	Enriching Word Vectors with Subword Information [71] Bag of Tricks for Efficient Text Classification [72]	Extension of Word2Vec considering sub-word level information (char n-grams)

			FastText.zip: Compressing text classification models [73]	
GloVe	Word embedding (hybrid approach)	2014	GloVe: Global Vectors for Word Representation [19]	Combine the best of count-based and predictive approaches

Table 6 - Summary of document and word embeddings techniques

During the last five years (from 2013 to 2018) *Word2Vec*, *FastText* and *GloVe* have been the most popular and widely used techniques for word embeddings in the *NLP* field, presenting state-of the-art results in this subject. Although not included in *Table 6* or in our analysis, by the end of 2018, once this work had been completed, *ELMo [93]* and *BERT [94]* were published and presented as new promising techniques for contextual word embeddings. We encourage the reader to check them for a full picture of the current state-of-the-art techniques regarding word embeddings.

In the next Chapter, a novel approach for building a vector space model for Internet domain names will be presented. Looking for computational efficient algorithms that can scale to large corpus at the internet scale, the techniques highlighted in bold in *Table 6* are evaluated. These techniques are based on previous architectures for language models, being *Word2Vec* an evolution (a smart simplification and optimization) of the work described by *Yoshua Bengio* et al. in [50]. *App2Vec* although not being originally designed for word embeddings (it was created for mobile apps embeddings) is an extension of the *CBOW* architecture of *Word2Vec* (a weighted version based on time gaps) and it has been employed in similar scenarios, therefore it is interesting to see how well it can generalize to our specific problem. Finally, the last technique that will be evaluated is *FastText* which is also an extension of *Word2Vec* but adding information to the sub-word level (character n-grams). As we will see in the next chapter, capturing information to the subword level is particularly helpful in our scenario where morphological information of Internet domain names gives important insights about their semantic.

Part II
DNS Vector Space Model

4. Chapter Ⅳ - Building the DNS-VSM

In Section 2.2, the importance of having methods and metrics for measuring semantic similarities between domain names was discussed. Also, it was noticed that having such kind of metrics is the key issue and the most fundamental requirement for creating numerous applications in different areas like user experience (recommend similar sites based on previously visited sites), security (filter of inadequate or risky content), data analysis (clustering, anomaly detection, etc), strategic competitiveness (identify competitors), performance optimization (cache, response time, etc), among others.

In this chapter, we will present a novel approach for building a vector space model for Internet domain names, that we will call *DNS-VSM*. This new approach has as main goal to solve the problem of finding semantic similarities between domain names in an efficient way without suffering of all the disadvantages of the current approaches already described in Section 2.2.5. The main advantages of the proposed solution are:

- It is not intrusive, and it is totally transparent for Internet users (users do not need to install anything).
- It does not require to know anything about the kind of content hosted in those domains.
- It does not require to trust in third parties.
- It does not need to offer any service to motivate users to install something.
- It works well no matter the user's device.
- Since it is a centralized approach, it is easy to develop/maintain/update.

- It is not restricted to any specific Internet user segment.

- Its design and structure are based on a vector space model, which allows to:

    - Easily find similar domain names (just by calculating the neighbors vectors in the space, using some distance metric between vectors).

    - Discover more complex relationships between domain names by applying basic math operations (addition and subtraction) with the vector representations of the domain names.

In order to achieve these goals, different word embedding techniques from the ones studied in Chapter 3 are applied to a corpus that is built from millions of anonymized *DNS queries*. These queries are taken from numerous log files saved by *recursive DNS servers* (see Section 2.1) that are owned by one of the most important Internet service providers in Uruguay. This approach is the first one (at least to the best of our knowledge) that presents a publicly available solution that makes use of *DNS* traces for finding semantic similarity between domain names through the usage of techniques that are taken from the *NLP* field.

As we will see during this chapter, the *DNS-VSM* will allow us to find semantic and syntactic relationships between domain names only by analyzing *DNS* traces of users, without requiring any previous knowledge about the domain's content itself and without requiring users to install anything. Other interesting aspect about the *DNS-VSM* is that its design and hidden linear structure will be helpful for building linear combinations between vectors, and keeping meaningful information as result.

The rest of this chapter is organized as follows: firstly in Section 4.1 a descriptive analysis of the raw data used as the initial input for learning DNS embeddings is shown. Then, in Section 4.2 the preprocessing steps that transform this raw data into the text corpus required by the different unsupervised learning techniques that are evaluated is described. The evaluation and comparison of these techniques are executed using the evaluation framework presented in Section 4.3 where also the baseline

models are introduced. In Section 4.4 the first model based on *Word2Vec* (see Section 3.2.2.3) is described in details. It's worth mentioning that most of the experiments and results obtained from this first model were summarized in publication [95]. In Section 4.5 the addition of the time factor (the elapsed time gap between two consecutives *DNS* queries requested by a same user) is studied and a second model based on *App2Vec* (see Section 3.2.2.4) is evaluated. A final improvement by considering sub-word information is described in Section 4.6 where the last model based on *FastText* (see Section 3.2.2.5) is presented. Finally, a summary analyzing the results and discussing possible use cases for the *DNS-VSM* is presented in Section 4.7

.

4.1. Descriptive analysis of the data

For this research, the data was provided by a large Internet Service Provider (*ISP*) from Uruguay. This ISP has millions of Internet users, and many recursive *DNS* servers. Everytime a user's internet browser needs to resolve a domain name a *DNS* query is requested to some of these servers as we saw in Section 2.1. The reader should notice that no additional software executes at the client side, requests to *DNS* servers are triggered in background while users browse Internet using any kind of device (desktop, mobile, etc).

In particular for this study, anonymized *DNS* queries saved in log files by the *DNS* servers were analyzed. The log files contain information from 11 different days in the period December 2012 - March 2013. Each line of log data shows the date and time of the *DNS* query, the anonymized IP address of the client, the domain name that has been requested and the type of *DNS* query. *Figure 19* shows an example of consecutive *DNS* queries requested by the same anonymized IP address. In the following, the expression "*user trace"* will refer to all the consecutive *DNS* queries requested by the same IP in some period of time (the time window*)*.

21-Mar-2013 06:06:47.2 client <anonymized ip>: query: apps.facebook.com IN A
21-Mar-2013 06:06:48.1 client <anonymized ip>: query: profile.ak.fbcdn.net IN A
21-Mar-2013 06:06:53.2 client <anonymized ip>: query: pixel.facebook.com IN A
21-Mar-2013 06:08:10.7 client <anonymized ip>: query: jacaranda.ceibal.edu.uy IN A

Figure 19 - Example of consecutive *DNS* queries

As we will see in section 4.2 the first step in the preprocessing phase is to filter *DNS* log entries based on the record type (only types A and AAAA are kept). Hence, once we have the log files filtered by record type, we can formally define a *DNS* log file as a sequence of $<IP_i , d_j , t_k>$, where the client IP_i queried the domain d_j at time t_k.
In the log there is a set of unique (anonymized) IPs representing each client c ∈ C, and for each client we have a DNS trace $t_c \in T$, that is a sequence $t_c = \{< d_1, t_1 >, <d_2, t_2 >, . . ., <d_n, t_n >\}$. The problem to solve is to learn a similarity function $sim(d_i , d_j)$ between any two domain names d_i and d_j , using only the set of traces T.
This problem is very similar to find semantically similar words, where a trace of domain names can be mapped to a sequence of words in a sentence. Following a similar reasoning (and although it is out of the scope for this research) the set of *DNS* queries that corresponds to a same IP could be mapped to a set of paragraphs in a document, and then a user's navigation profile associated to that set of *DNS* queries could be mapped to the concept of document, allowing in this way to apply techniques and methods for document embeddings (see Chapter 3) to get vector representations for user profiles.

All these DNS queries are saved in the *DNS* servers log files as compressed .bz files. The amount of data collected in just one day is extremely large, containing more than 3.5 billions of queries, 58 millions of unique domain names, and 550 thousands of unique IPs. Data across different days are pretty similar, with a little increment with time, as shown in *Figure 20* for 3 different days. As we can see, A and AAAA record types correspond to more than 90% of the queries in an average day (*Figure 21*), and other types are less relevant when studying user's web navigation habits. For this reason, during the preprocessing phase, data is filtered to keep only these kind of queries (A and AAAA). For these record types we can see a minimum between 5 and 6 a.m. and maximum between 8 and 10 p.m. approx as it is shown in *Figure 22* and *Figure 23* for the same three days.

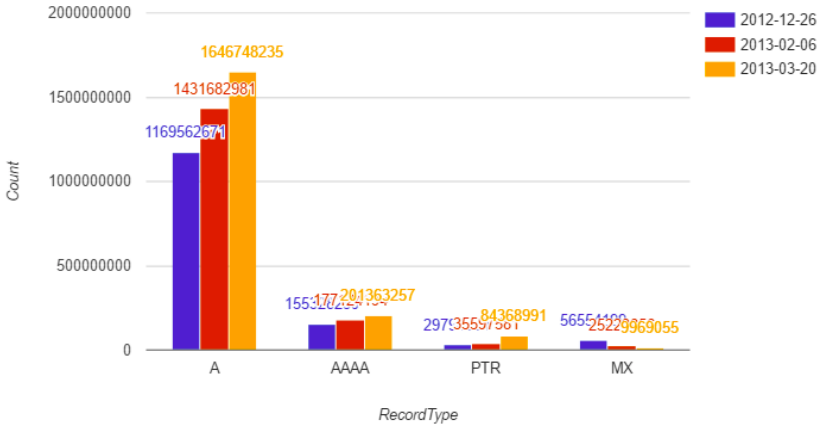


Figure 20 - Number of DNS queries for main records types on 3 different days

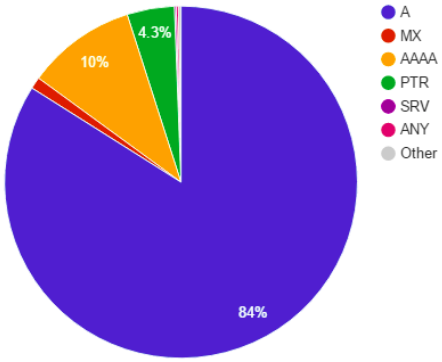


Figure 21 - Distribution of DNS queries types on March 20 of 2013

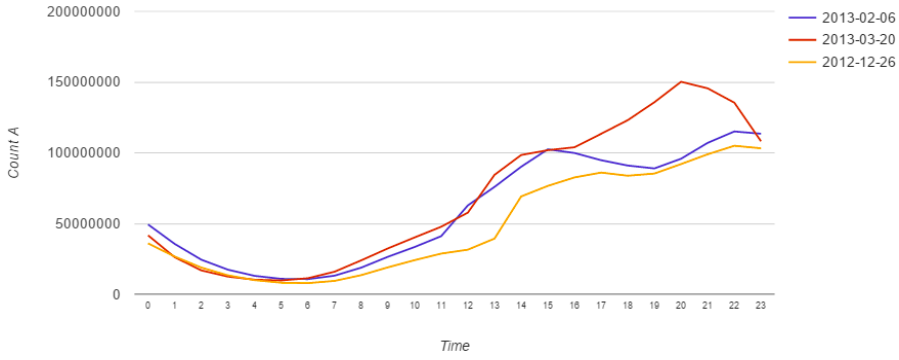


Figure 22 - Number of type A DNS queries per hour on 3 different days

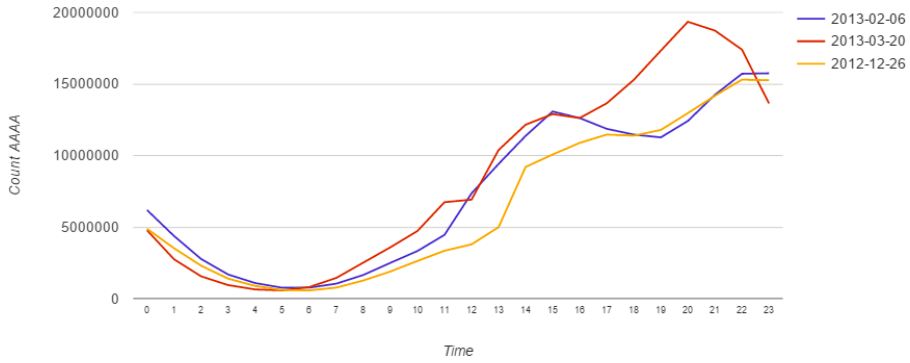


Figure 23 - Number of type AAAA DNS queries per hour on 3 different days

4.2. Preprocessing phase

Raw data obtained from the DNS log files are not exactly in the form that we need. For this reason, raw data need to be preprocessed, cleaned and transformed before using them as input for training the different machine learning models that we will be evaluating.

There are some characteristics and limitations about the data collected and how the DNS system works that are important to understand before moving to the specific preprocessing data pipeline that was applied. The preprocessing steps that will be detailed later, in some way or another, attempt to mitigate these issues or to simplify some of these problems. The following list summarizes the main issues that we need to face:

A. When studying user's web navigation habits, A and AAAA record types are the most important, but **other types like MX, PTR or SRV are also present in the logs and they can be considered as noise** when trying to understand domain names similarities based on click-streams (based on how users browse the web).

B. Since DNS resolvers clients typically cache the requests, we do not have information about how often a domain is visited. We only have one request after the domain cache times out and then it is cached

again. Therefore, **the DNS traces are not a good source to measure the period of usage of a domain, just the first access**.

C. Also, **NAT enabled gateways hide the activity of many users behind a single public IP address**. This is very common in enterprise and residential connections, but not in mobile services. It means that **an IP's trace can mix multiple clients, in some cases thousands of them**. Usually the ISP assigns disjoint range of IPs to each kind of service, therefore it is possible to separate enterprise, residential and mobile traces.

D. It is a good practice of ISPs to assign a dynamic IP address to each service, where the client needs to be reconnected after a fixed period (for example 12 hours) and a new IP is assigned. Therefore, an IP identifies a particular service/client in a period. **The larger the period, the more likely that the IP would have been reassigned to another client**.

E. Internet Content Providers (ICPs) typically use several sub-domains in order to provide their content, moreover they usually use external services to provide part of their content (for example they use content delivery networks to provide static and video content). Therefore, **when a client access to a service, it usually adds several subdomains and also other domains to the trace**. This is very consistent between different clients that access to the same service, but it is not a simple task to extract the knowledge of the service used by the client from this trace.

F. In a similar way, **there are applications in client's devices (mainly in mobile devices) that generates traffic in background** (and therefore queries in the trace) **without an explicit action of the user** (for example antivirus, email clients, etc.). These queries are mixed in the traces and they can act as fingerprints for identifying similar traces from a particular machine but are not really helpful when trying to understand relationships between domain names by analyzing how users browse the web.

Hence, in order to mitigate the impact of these limitations in our procedure, we propose several preprocessing steps (filters) that are executed sequentially in a preprocessing execution pipeline.
The input to this pipeline is a set of DNS servers log files and the output is a single big text file that we will refer as "*the corpus*" in the rest of this document. This corpus is the unstructured data that we will be used as input for the different models under evaluation.
It's worth mentioning that a second version of the corpus adding time gaps between domain names was also generated specifically for evaluating the *App2Vec-like* model. Details about this second version of the corpus is given in Section 4.5 when describing the *App2Vec-like* model applied to the *DNS* corpus.

The preprocessing pipeline:

1. **DNS record type filter**
   As it is shown in *Figure 21* the A and AAAA record types correspond to more than 90% of the queries in an average day. During the preprocessing phase, data is filtered to keep only these kind of queries (A and AAAA). By doing this we focus exclusively in that kind of record types that are generally used by *DNS* resolvers when asking for internet domain names resolution during user's web navigation sessions, thereby minimizing the effect of issue A. The reader should notice that the amount of data removed by discarding other kind of *DNS* record types does not affect substantially the remaining dataset.

2. **Service type filter**
   A subset of the data is used, considering queries that are requested by IPs that belong to some known ranges corresponding to residential or mobile services and discarding enterprise services that are used by companies with many employees browsing the web behind the same IP. Although residencial connections potentially could allow more than one device connected at the same time, we decided to include them knowing that in some cases these residential IPs could contain data from more than one device or user, but we hope that our solution can deal with this noise by using a big amount of data.

The reader should notice that this is a simplification in order to minimize the amount of devices connected with the same IP, thereby reducing the possibility of having traces that merge queries from the same IP but from different devices or users (issue C).
In Section 2.2.5 we pointed out that one disadvantage of the current approaches that make use of client side components (add-ons, toolbars, etc) is that they do not work well on mobile devices' browsers. Hence, by including mobile IP ranges in our traces we are solving both, the problem that current approaches suffer regarding mobile devices as well as the problem of possible multiple devices/users behind a same IP (issue C).

3. **Simplification of subdomain component**
   This is a simplification aiming to help in solving issue E regarding subdomains. Generally websites include several subdomains in order to improve load time by loading in parallel content from many sources (static content like javascript/css files, or multimedia assets, webservices endpoints, mirror backend servers, etc). This step in the pipeline truncates the last labels of a domain name in the following way: if the top label is a country code (*ccTLD*[43]) then the domain is truncated to the first 3 levels, else (*gTLD*[44]) the domain name is truncated to the first 2 levels. Please see Section 2.1 for a deeper explanation about the hierarchy structure of domain names.

4. **Removal of top queried domains and well-known applications domain names**
   Top domains like *google, facebook, youtube, root-servers* (among others) that are queried all the time in any context do not give us any relevant value.
   Hence, as part of the preprocessing pipeline these domain names are included in a fixed black-list, and excluded from the corpus. The idea behind this is similar to the one used when processing natural language data in text, where a set of words (called common or stop words like *the, is, at, which*, etc) are filtered before processing the text because they are not really important (they do not add any

[43] https://icannwiki.org/Country_code_top-level_domain
[44] https://icannwiki.org/Generic_top-level_domain

meaningful information). This workaround also helps in solving issue F. The following list of domain names was excluded during this filter:

ZXV10\\032H108, avast, facebook, fbcdn, youtube, flickr, avira, twitter, ytimg, akamaihd, akamai, akamaiedge, microsoft, msftncsi, anteldata, bing, eset, avg, avira-update, yahoo, mozilla, doubleclick, google, gstatic, google-analytics, googleusercontent, googleapis, googlesyndication, googleadservices, dyndns, antel.net, root-servers, windowsupdate, no-ip, changeip, whatsapp, kaspersky, live, msn, scorecardresearch, skype, verisign, nod32.

5. **IP grouping**

   It is important for our algorithms that consecutive DNS queries in the corpus file are part of the same user's web browsing session. But DNS servers receive tons of DNS queries from multiples users at the same time and thus, DNS queries appear merged all together in the log files making difficult to visualize activity by IP. For this reason, DNS queries are re-arranged sequentially in time grouped by IP.

   For example the following input:

   21-Mar-2013 06:06:47.2 client <anonymized ip1>: query: domain1.com IN A
   21-Mar-2013 06:06:48.1 client <anonymized ip2>: query: domain2.com IN A
   21-Mar-2013 06:06:53.2 client <anonymized ip3>: query: domain3.com IN A
   21-Mar-2013 06:08:10.7 client <anonymized ip2>: query: domain4.com IN A
   21-Mar-2013 06:08:11.2 client <anonymized ip3>: query: domain5.com IN A
   21-Mar-2013 06:08:11.5 client <anonymized ip1>: query: domain6.com IN A

   Is re-organized in this way:

   21-Mar-2013 06:06:47.2 client <anonymized ip1>: query: domain1.com IN A
   21-Mar-2013 06:08:11.5 client <anonymized ip1>: query: domain6.com IN A
   21-Mar-2013 06:06:48.1 client <anonymized ip2>: query: domain2.com IN A
   21-Mar-2013 06:08:10.7 client <anonymized ip2>: query: domain4.com IN A
   21-Mar-2013 06:06:53.2 client <anonymized ip3>: query: domain3.com IN A
   21-Mar-2013 06:08:11.2 client <anonymized ip3>: query: domain5.com IN A

And the final corpus considering just sequences of domain names will be:

domain1 domain6.
domain2 domain4.
domain3 domain5.

The reader should notice that IP's values are not included in the final corpus, but after the preprocessing pipeline is completed, we are guaranteed that domain names that appear in the same sentence (a sentence is a sequence of domain names separated by white space and a final dot marking the end of the sentence) correspond to DNS queries (A or AAAA) requested by the same IP (residential or mobile IP).

6. **Removal of automatic requests (that are not click-streams)**
   Other problem that we can see in issue *E* is related to those contents that are loaded automatically from the same or other domain name as well as automatic redirections (among others). Since we want to study relationships between domain names according to how they are consumed by users, we want to eliminate any request that is not explicitly executed as response for an Internet user's action (for example the list of resources automatically loaded after accessing a web page).
   Redirections and references to content hosted in external sites could be detected easily in html pages, but it can be very expensive and time consuming in a big dataset. Hence in our case, we will get that information based on data evidence. In order to identify these domains we apply the following empirical rule: domain names with at least 100 occurrences, and that 90% of the time (or more) are queried immediately after a previous domain (3 seconds window) are added to a blacklist (csv file) and excluded from the final corpus. After applying this filter, 5345 domain names were added to the blacklist, therefore removed from our dataset.

7. **Simplification in the navigation path**
   In text documents the same word does not appear repeated many consecutive times. Also, when studying DNS similarities based on navigation contexts, that a domain is similar to itself does not add any relevant information. For this reason, we simplify the navigation path by removing consecutives occurrences of the same domain name and leaving just one of them. For example, a sequence like

d_1->d_2->d_2->d_2->d_3->d_3->d_4 is simplified to d_1->d_2->d_3->d_4. This simplification will help also in reducing the corpus size.

8. **Split of long traces (using time window)**
   As we saw in Section 3.2.2.3, an important hyper-parameter in *Word2vec* is the size of the context window, and this parameter is also used in *App2Vec* (see Section 3.2.2.4) and *FastText* (see Section 3.2.2.5) as well. Thus, since the word embedding techniques that we will be evaluating consider that two consecutives words are always part of the same context, we want to preprocess the traces in same way that we can ensure that two consecutives domains in the output corpus are effectively part of the same user's web navigation context.
   In order to meet this requirement, we map the concept of context window to time window and we break long traces for a same IP into many shorter traces (5 minutes) that form sentences in the corpus. Each sentence is composed by a set of domain names requested inside a same time window (same navigation context) by the same IP.
   It's worth mentioning that this step in the preprocessing pipeline is not applied when building the corpus version for evaluating the *App2Vec-like* model. In the particular case of *App2Vec*, traces are splitted only if they are longer than 12 hours because of the dynamic IP characteristic of *DNS* providers for residential IPs. For traces lower than 12 hours we do not apply any break and we leave the time factor added by *App2Vec* to deal with this problem, hoping that two consecutives domain names that are far in time are weighted lowly and thus adding an insignificant component to the similarity.

Table 7 summarizes the main challenges that we faced during the preprocessing phase and the workarounds applied in order to solve them.

After preprocessing the data, the final input is a sequence of 53 millions of domains, where the unique domains are 1.4 million. As expected, there is a large variation of popularity between domains. *Figure 24* and *Figure 25* show the cumulative percentages of requests per domain, where domains are represented by popularity position (from left to right) on the x-axis. We can see that top 5 thousand domains accumulate

75% of the total queries, top 798 domains accumulate the 60% and top 10 domains accumulate the 10%. Due to the logarithmic characteristic of the function, working with a reduced vocabulary but very representative of the total traffic was seen as a good alternative in order to optimize the execution time for our experiments.

Problem	***Solution***
A (not all *DNS* record types are important when studying web navigation habits)	1 (*DNS* record type filter)
B (*DNS* caching)	The time gap between two consecutive domains after applying solution 6 (*removal of automatic requests that are not click-streams*) and 7 (*simplification in the navigation path*) can be used as an approximation to the period of usage of a domain. When evaluating the *App2Vec-like* model in our *DNS* corpus, this time gap will be used as distance to weight how near in the web session two domains are, knowing that it could be not totally exactly (because of the *DNS* caching that we cannot bypass). This, probably could be one of the reasons why the results for the *App2Vec-like* model will not be as good as we can expect in advance.
C (multiple users using the same IP)	2 (service type filter)
D (dynamic IPs)	8 (split of long traces using time window)
E_1 (subdomains typically do not add to much value)	3 (simplification of subdomain component)
E_2 (domains that are automatically requested without any user's action)	6 (removal of automatic requests that are not click-streams). Also, 4 (removal of top queried domains and well-known applications domain names) helps by removing requests that are triggered automatically by antivirus software or similar software in user's devices.

Table 7 - Main problems faced during the preprocessing phase and the proposed solutions

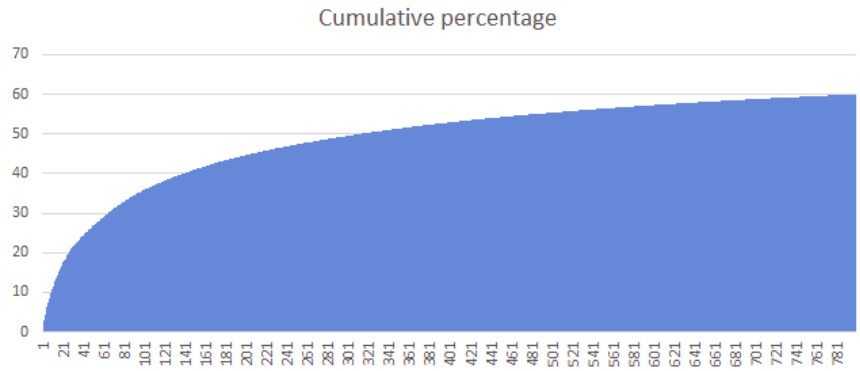


Figure 24 - 60% cumulative percentage of requests per domain.

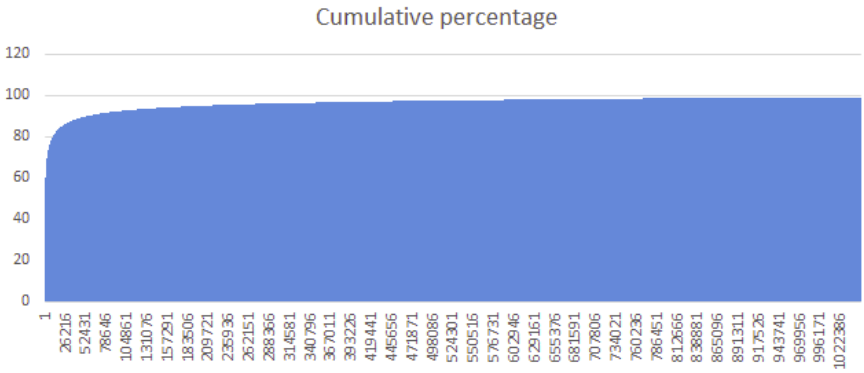


Figure 25 - 100% cumulative percentage of requests per domain.

When we studied the optimization tricks proposed in order to scale *Word2vec* (see Section 3.2.2.3) one of them consisted in reducing the number of neurons in the output layer. The size of the output layer impacts considerably in the algorithm performance due to the computation of the final softmax classification layer.

For this reason, the trick proposes to remove all rare words from the vocabulary (words with a number of occurrences lower than a threshold) and then add a new single token to the vocabulary as a representative class for rare words. By doing this, the output layer can reduce its size considerably, getting an important performance enhancement.

Motivated by this trick, and since the number of neurons for the output layer in our work is mapped directly to the number of domain names, we decided to work with a vocabulary built from the top 40 thousand most popular domains. It's worth mentioning that we do not remove any domain name from the dataset explicitly, but we leverage the *Python's* interfaces

for the word embeddings algorithms (in both *Tensorflow*[45] and *Gensim*[46]) to specify the number of words that we want to keep in the vocabulary.
These top 40k domains represent 88% approximately of the total requests under study. Furthermore, an *unknown token* (the *UNK* token) for rare domains was added to the vocabulary. By pruning the input to the top 40k domains, we can work effectively with a very good representative subsample of the dataset and allowing the execution of the different algorithms efficiently.
By the end of the preprocessing phase, a depurated and reduced set of *DNS* traces is generated and it is ready to be used as input for training our word embedding models. In the end, these models will give us the vector representation of domain names and thus, our *DNS* vector space model.
But before moving to the details regarding the word embedding models, the evaluation framework that will be used for comparing the results of the different models is presented in the next section.

4.3. Evaluation framework

One of the main challenges that we need to address is to evaluate the quality of the semantic relationships between domain names that are discovered by our unsupervised process.
When working with supervised learning either on a regression or classification problem, the true labels (responses) are well known in advance and they can be used to compare against the predicted values to measure how good a model is. But in our case, there are not true labels, hence it is very difficult to know whether our models are learning good representation for domain names or not.
In order to solve this problem, and motivated by our previous research about similar works, in this research we use the service called *similar sites* [47] (a feature offered by *Alexa*[48] in its *Audience Overlap Tool*[49]) as reference for comparison and, as a trusted source (ground truth) for evaluating the quality of our results.

[45] https://www.tensorflow.org/
[46] https://radimrehurek.com/gensim/
[47] https://www.alexa.com/find-similar-sites
[48] https://www.alexa.com/
[49] https://try.alexa.com/marketing-stack/audience-overlap-tool

This service allows to query for a specific domain name in order to know possible competitors based on considering similar sites. In its paid version, this service retrieves the top 100 most similar sites for a specific input domain name (more details about *Alexa* and its tools can be found in Section 2.2.1).

As part of this work, we built a tool that given the list of the top used domains in our vocabulary, it retrieves the similar sites from *Alexa*. More precisely, we queried for 15000 top domains in our vocabulary, for which we were able to find a matching in *Alexa* for 14490 domains (96.6%). The reason why not all of our domains were found in *Alexa* could be one of many causes, for instance because not all of them are Internet web sites or because our dataset contains domains that do not exist any more, among others reasons. Also, of those 14490 domains, a total of 11426 have at least one similar that also belongs to our dataset, therefore they are the ones that are taken to form our evaluation set.

The result list that we get from the *Alexa*'s service has a default order by an *Overlap Score value*. The meaning of this value according to *Alexa* is the relative level of visitor (audience) overlap between any site and the target site. A site with a higher score shows higher audience overlap than a site with a lower score. This order will be important when comparing the results from our models versus the most similar sites retrieved from *Alexa*.

It is not trivial to compare results from different models that learn similarities between domain names, because it implies to have a metric that compares two ordered list of similar domains for each domain considered in the vocabulary. The simplest metric would be to compare just the most similar domain, considering a success only when it is the same than the most similar site (top 1) in *Alexa*'s response, and averaging the results between all the domains in the vocabulary. As we will see, this is a particular case of the *Mean Average Precision (MAP)* metric [96], which can also compare the first *k* similar domains (not just the first one).
MAP metric is the mean of the *Average Precision (AP)* computed for each element (in our case, for each domain). Some of the characteristics that we were looking and the *AP* metric has are:

- It assigns values between 0 and 1.

- If both lists of similarities are identical, then its result is 1.

- If both lists of similarities are disjoint, that is, if none of the domains in our predicted similarity list are present in *Alexa*'s list, then its result is 0.

- Order in predicted list matters. Suppose that for a given domain name, a first model computed the list of its *k* most similar domains, and the top half of this list also appears in the similar sites retrieved from *Alexa*. Now, suppose that we have a second model that computes a second similarity list where the matching with *Alexa* appears only in the last half of the list. Despite for both lists we have matched *k* / 2 domains with *Alexa*, we want that the metric, when evaluating the first list to be higher than for the second list. That is because the *k* / 2 matched domains for the first list are located above in the ordered list, which indicates that similarities for those domains are more accurate than the similarities for the *k* / 2 matches in the second list.

The *AP@k* (the average precision considering the top *k* elements) and the *MAP@k* (the mean of *AP@k* computed for all items) are metrics that have been widely used in the information retrieval field, as well as in the recommender systems domain. Before moving to the formulas, let's illustrate its behaviour through an easy example.

Suppose user *U* likes two movies M_a and M_b, then if recommender *R1* recommends movies $[M_c, M_a]$ (ordered list), this is not as good as the recommendations given by a second recommender *R2* that recommends movies $[M_a, M_c]$ (ordered list). If we compute the *AP@2* metric (*AP* metric considering the top two items) for user *U*, despite both recommender systems recommend the same set of items, since the order in the recommendation list is relevant and it is taken in consideration by the *AP* metric, then we will get that:

$$AP@2(up, recs1) < AP@2(up, recs2) , \qquad (Eq.\ 32)$$

where *up* = $\{M_a, M_b\}$ (set of preferences of user *U*), *recs1* = $[M_c, M_a]$ (ordered list of recommendations given by recommender *R1)*, *recs2* = $[M_a, M_c]$ (ordered list of recommendations given by recommender *R2*). Therefore, from this user's point of view, recommender *R2* is better than recommender *R1*. We can repeat the same reasoning for every user in the system and then aggregate the results by taking the mean of all *AP@2* in order to get a global performance score, and that is what *MAP@2* is.

If we map this example in the recommender systems domain to our escenario, then we can think in the following associations:

- The most similar sites retrieved from *Alexa*'s similarity service for a given domain *d*, corresponds to the set of preferences of user *U* (the ground truth)

- The most similar sites to *d* (ordered list) found by *Model1* (first word embedding model under evaluation) corresponds to the ordered list of recommendations given by recommender *R1* for user *U*

- The most similar sites to *d* (ordered list) found by *Model2* (a second word embedding model) corresponds to the ordered list of recommendations given by recommender *R2* for user *U*

Hence, if we compute *AP@k* considering the top *k* similar sites to *d* found by the two models *Model1* and *Model2* using *Alexa*'s response as the ground truth, then we can get a good criteria for comparing both models, being *Model1* better than *Model2* (computing similarities to *d*) if and only if:

$$AP@k(model1,\ d)\ >\ AP@k(model2,\ d)\,. \qquad (Eq.\ 33)$$

And then, *MAP@k* metric can be computed by averaging *AP@k*, considering every domain names in our evaluation set (not only *d*). By doing this, we can get a global performance score for our models, being *Model1* better than *Model2* if and only if:

$$MAP@k(model1)\ >\ MAP@k(model2)\,. \qquad (Eq.\ 34)$$

A visual inspection of the results that come from *Alexa* clearly shows that only the first domains are relevant followed by popular or generic domains. Therefore, we define a new variable in our evaluation framework that we call k_{actual} and we consider only the first k_{actual} domains in the *Alexa*'s response that are also present in our data. We ignore others domains in *Alexa*'s response since they will never show up in our predictions.
This new variable, will help us to mitigate the lack of the *MAP* metric related to the order in the *actual* list (order in the similar sites retrieved from *Alexa*, does not matter). The metric uses the *Alexa*'s list as a set without any particular order, where it is the same to have a coincidence with the first or the last domain in the set. We will mitigate this problem varying the k_{actual} value, and evaluating the impact into the *MAP@k* value.

That said, now we present the formulas for the *AP@k* and *MAP@k* metric. We have a set of domains in our vocabulary, $d \, \varepsilon \, D$, for which we know the *actual* ordered list of top k_{actual} similar domains from *Alexa*, and an ordered list of predicted top *k* similar domains in our vector space model. In order to compare these two lists of (*actual* and *predicted*) similar domains for a specific domain *d*, we use the *Average Precision (AP)* over all possible *recall* values:

$$AP@k|_d = \sum_{n=1}^{k} P(n)\Delta(n) \, , \qquad \text{(Eq. 35)}$$

where *n* is a position in the predicted list of similar domains to *d*, *k* is the size of the predicted list, $P(n)$ is the precision considering only the first *n* domains in the list, and $\Delta r(n)$ is the change in *recall* from domains *n - 1* to *n*. *Precision* and *Recall* metrics come from information retrieval theory, see their definition in [96].

With the *Mean Average Precision (MAP)*, we summarize the comparison of *actual* and *predicted* lists for all available domains. Given a set *D* of domain names, the *MAP@k* metric is computed as the mean of the average precision scores:

$$MAP@k = \frac{\sum_{d \in D} AP@k|_d}{|D|} \ , \qquad \text{(Eq. 36)}$$

where $d \, \varepsilon \, D$ is a domain for which we know the top k_{actual} *actual* similar domains from *Alexa*, and an ordered list of predicted top *k* similar domains in our vector space model. The reader can refer to [96] to see a detailed explanation about *mean average precision* and the formula above.

Besides the *MAP@k* metric, the *precision* and *recall* metrics used for evaluating classification models in the machine learning field have been used, as well as the *f1-score* which is the harmonic mean between them.
In order to use these metrics, we consider each domain name as a different *class* (a.k.a *tag* or *label*) and our model as a *multi-label classification* [97] model (the output of the model is a set of labels that correspond to the similar domain names computed by the model for a given input).
Hence, given a domain name *d*, if we denote $P_d = \{p_1, p_2, \ldots, p_n\}$ as the set of predicted most similar domain names to *d*, $T_d = \{t_1, t_2, \ldots, t_{kactual}\}$ as the ground truth (the set of domain names retrieved from the *Alexa*'s service considered as the true/correct most similar domain names to *d*), then *Equation 37*, *Equation 38* and *Equation 39* describe how the *precision*, *recall* and *f1-score* are computed respectively.

$$Precision_d = \frac{|T_d \cap P_d|}{|P_d|} \ . \qquad \text{(Eq. 37)}$$

$$Recall_d = \frac{|T_d \cap P_d|}{|T_d|} \ . \qquad \text{(Eq. 38)}$$

$$F1_d = 2 \cdot \frac{precision_d \cdot recall_d}{precision_d + recall_d} \ . \qquad \text{(Eq. 39)}$$

Finally, as shown in *Equation 40*, *Equation 41* and *Equation 42*, for each model *m* that we want to evaluate, we can measure its performance by computing the average metrics considering each domain name *d* in our vocabulary *D* and the most similar domain names to *d* given by *m*.

$$Precision_m = \frac{\sum_{d \in D} precision_d}{|D|} \quad . \qquad \text{(Eq. 40)}$$

$$Recall_m = \frac{\sum_{d \in D} recall_d}{|D|} \quad . \qquad \text{(Eq. 41)}$$

$$F1_m = \frac{\sum_{d \in D} F1_d}{|D|} \quad . \qquad \text{(Eq. 42)}$$

4.3.1. Baseline models

Two baseline models were implemented in order to have a baseline to compare the performance of the new models that will be evaluated.
Firstly, a *random guessing* baseline model was implemented. This first model returns *k* random domains as the most similar domains to a given domain name.
Next, an extension of the *majority class* baseline model was implemented based on popularity and returning always the top *k* most popular domains as the most similar domains to any other domain.
Finally, an hybrid approach using a weighted schema by returning random domains with probability weighted by popularity was implemented but since it did not perform better than just the plain popularity based model, it was not considered here as a different case.

The *MAP* metric (using $k = 3$, $k_{actual} = 3$) computed for our baseline models obtained a result of 0.00005 and 0.016246 for the *random guessing* and the *popularity based* models respectively.

4.4. Creating DNS embeddings using Word2Vec

When studying the theory of *Word2Vec* in Section 3.2.2.3, it was pointed out that *Word2Vec* is a family of algorithms for word embeddings generation that supports two main configurations of the underlying neural network structure: the *CBOW* and the *Skip-Gram* architectures. In this

section, the main details about the creation of our first *DNS* embedding model based on the *Skip-Gram* architecture of *Word2Vec* is presented. Latter in Section 4.5 an *App2Vec-like* model (see Section 3.2.2.4) based on the *CBOW* architecture is also evaluated.

As any other word embedding algorithm, we can think *Word2Vec* as a black box that receives a text as input and it generates a vector representation for each word in the text (a.k.a *embeddings*) as output. In Section 3.2.2.3 some of the main characteristics of the generated embeddings were discussed, in particular it was shown that the vector representation of semantically related words (that generally share similar contexts) are located nearby in the high dimensional vector space. Thereby, our first approach took advantage of that characteristic. After having applied all the pre-processing steps (see Section 4.2), the final corpus is used as the input text for the *Word2Vec* algorithm, which was implemented (in our first approach) using the *Tensorflow*[50] library for Python. Finally, in order to get the most similar domain names to any other domain name, the *cosine similarity* between vectors is computed to get the nearest neighbors vectors. The more similar their vector representations, more semantically related the domain names are.

When working with *Tensorflow*, a computational graph is firstly defined and then, an execution session is run using an instance of that graph. We based our implementation in [36], where a *Tensorflow* graph is used to model the *Skip-Gram* neural network architecture. As we saw in Section 3.2.2.3.1, the *Skip-Gram* model tries to predict context words from a target middle word. Hence, there are two input nodes in the graph, one for the target word and another one for the context word. During the training phase, many combinations of couples in the form *<target, context>* words are randomly subsampled and its integer representations are fed into the input nodes of the graph. The integer index for each word is assigned once before starting the training phase, while loading the vocabulary. In that moment, an integer value in the range *[0..n-1]* (where *n* is the vocabulary size) is assigned to each word and this value is used later as an alias when the word is fed into the network. The reader probably have noticed that we are not using the *one-hot encoding* representation of the words as inputs to the network (as it's supposed

[50] https://www.tensorflow.org/

according to the original *Skip-Gram* network architecture). This is because after adding the embedding layer to the graph (a uniform randomly initialized matrix to store the weights values associated to the connections between the input and the hidden projection layer that compose the shallow network architecture of the *Skip-Gram* model) we can leverage the *embedding_lookup*[51] operation provided natively by *Tensorflow* which is specifically designed and optimized for an in parallel look up execution, achieving the same goal than the original *one-hot encoding* representation: to efficiently select the corresponding row for the input word from the embedding matrix. As optimizer for minimizing the loss function, the *Stochastic Gradient Descent (SGD)* technique was used.

As part of the implementation, two variations of the suggested improvements to the original model presented in [61] were implemented: i) the subsampling of frequent words and ii) the usage of a lightweight objective function rather the classic softmax.
The removal of stop (common or frequent) words is used to remove noise from the data, that is, to remove words (domain names in our case) that do not add any relevant information. In our specific scenario, these domain names correspond to those domains that are requested all the time, therefore being present in most of the contexts with many different kind of other domains (they do not belong to any specific field) and then, getting vector representations that are not relevant for finding similarities or analogical reasoning. In Section 4.2 we can see that this is what one of the preprocessing steps exactly does during the preprocessing pipeline. In that step, a set of commonly requested domain names like *Google* or *Facebook* (among others) are filtered and removed from the final corpus.
In regards to the usage of a lightweight objective function as improvement, we did not use exactly the *Negative Sampling* loss function suggested as improvement and instead, we used the very similar *Noise Contrastive Estimation (NCE)* suggested by *Mnih et al*. in [69] for which *TensorFlow* has a handy helper function *nce_loss*[52].

As an additional performance improvement, when loading the vocabulary from the corpus, we decided to limit the vocabulary size to the top 40k domain names with the highest frequencies (88% approximately of

[51] https://www.tensorflow.org/api_docs/python/tf/nn/embedding_lookup
[52] https://www.tensorflow.org/api_docs/python/tf/nn/nce_loss

the total requests, see *Figure 25*) and to include an *unknown token* (the *UNK* token) for rare domains. By doing this, since the size of the output layer depends on the number of words in the vocabulary, we can reduce significantly the number of neurons in the output layer, and thus, train the model efficiently with a very good representative subsample of the dataset.

Finally, some minor modifications to the original *Tensorflow* code were added mostly for saving partial checkpoints and summary data for evaluating the loss value (a measure of the network's error) during the training phase with *Tensorboard*[53].
For visualizing the generated vectors, the *t-Distributed Stochastic Neighbor Embedding (t-SNE)* [98] dimensionality reduction technique was applied. This technique is widely used for visualization of high dimensional datasets in two or three dimensions, while retaining the local structure of the data [98]. *Figure 26* shows the projection in two dimensions of some vectors (a subset of the full dataset) taken from a partial checkpoint during the training phase.

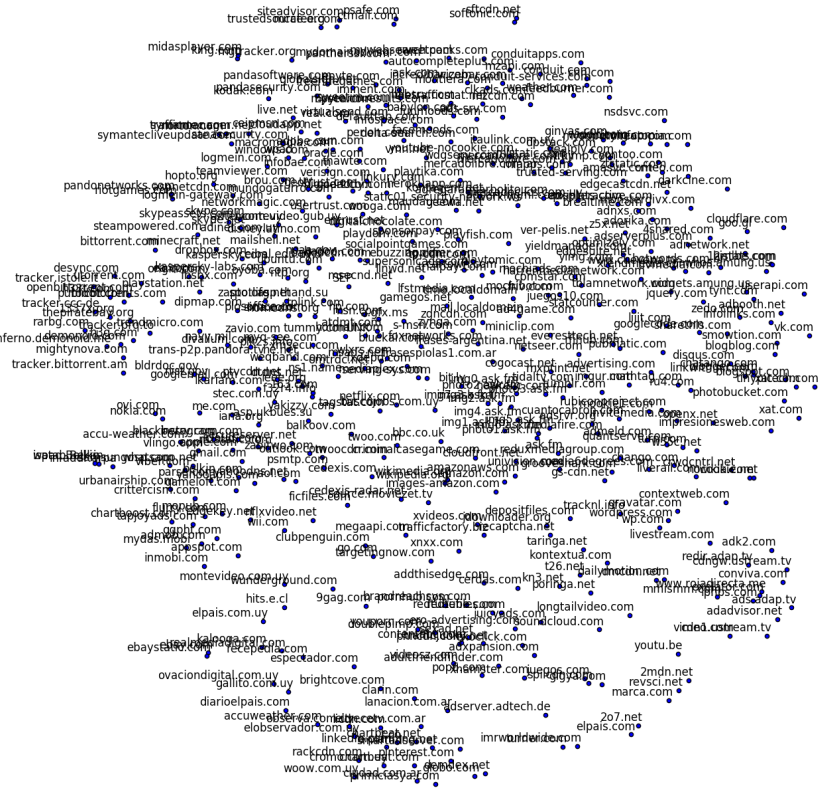


Figure 26 - Projection of domain names embeddings in 2 dimensions using t-SNE

[53] https://www.tensorflow.org/guide/summaries_and_tensorboard

Hyperparameter tuning

As most of the machine learning algorithms, the configuration of *Word2Vec* has a set of parameters that are not trainable and need to be configured in advance before running the training phase. This set of initial configurations to the model are called hyperparameters, and finding the optimal configuration is usually known as hyperparameter tuning, which is an important but time consuming phase of any machine learning project.
In particular, for the implementation of the *Skip-Gram* architecture in *Tensorflow*, the more important parameters that need to be configured are: the size of the vocabulary and the embeddings, the size of the skip window and batches, as well as the value of the learning rate.

The *embedding size* corresponds to the number of neurons in the hidden projection/embedding layer in the *Skip-Gram* architecture, and since the weights associated to the neurons in this layer are finally used as the values to represent the vectors for each word, this size implicitly define the number of dimensions that will be used to represent our domain names as vectors in the multi-dimensional DNS vector space. It is supposed that higher values of this hyperparameter would allow to discover more hidden or latent features by our learning algorithm. But a too high value of this hyperparameter could end up being expensive in resource usage, decreasing considerably the solution's performance and increasing the overall time for the learning process.
The *learning rate* is a well known hyperparameter used in neural networks to regulate the speed of the optimization process (when minimizing the cost/error function), generally done by the gradient descent technique. A low value of this hyperparameter can turn the learning process very slow. On the other hand, a high value of this hyperparameter could reduce the time for the learning phase (convergence time) but could also face the problem of not decreasing on every iteration (jumping over the minimum once and again repeatedly), having a solution that cannot converge. More details about learning optimization can be found at chapters 4, 5 and 8 in [26].
The *skip window* hyperparameter (a.k.a *"the context"*) is one of the most important variables in *Word2Vec*. It represents the size of the sliding window over the corpus and it indicates the number of domain names that

need to be considered at the left and right of a target domain when training the network.
Finally the batch size hyperparameter indicates the number of examples (domain names) that will form a new input set to present to the network for training.

Typically, in practice it's impossible to test all possible combinations of hyperparameters (we may have infinite options for some parameters). For this reason, we reduce the universe of possibilities to a subset of them, choosing a set of possible values taking suggestions from the corresponding documentation or in some cases just by intuition. Once all the possible values for each hyperparameter are specified, a *grid search* method considering all the possible combinations of these values is executed to find the best configuration for our *Skip-Gram* model. The different hyperparameter options that were tested are:

- Embedding size (dimension of the embedding vector): {8, 16, 32, 64, 128}
- Skip Window size: {1, 3, 5, 7, 10}
- Batch size: (170, 510, 1190, 1700)
- Initial Learning rate: {0.3, 0.5, 0.8}
- Vocabulary size: {40000}

As explained before, limiting the size of the vocabulary helps to improve the execution time and after studying the characteristic of our data we decided to use the top 40k domain names with the highest frequencies. Hence, during the hyperparameter tuning we used the fixed value of 40000 for the vocabulary size setting.

In order to compare the different settings for the hyperparameters, a new instance of the model is created for each configuration and then, the *MAP@k* metric (see Section 4.3) is evaluated for the particular case when $k = 1$ and $k_{actual} = 1$ (comparing just the most similar domain name returned by the model vs the most similar domain name retrieved from the *Alexa*'s service).
The best result was obtained using an embedding size of 128, learning rate of 0.5, window size of 3 and a batch size of 510. Nevertheless, as it is shown in *Figure 27* (each bar color represents a embedding size and learning rate combination) we also found very good results using an

embedding size of 64, learning rate 0.3 and same window size and batch size (3, 510).

When using an embedding size of 64 and 32, the results were within 99.9% and 99.5% (respectively) of the results obtained with the bigger embedding but the memory requirements are notably lower. This is the reason why in the end of this tuning phase we decided to use an embedding size of 64 dimensions. That said, in next sections if nothing different is indicated then we will assume that we are using embeddings with 64 dimensions.

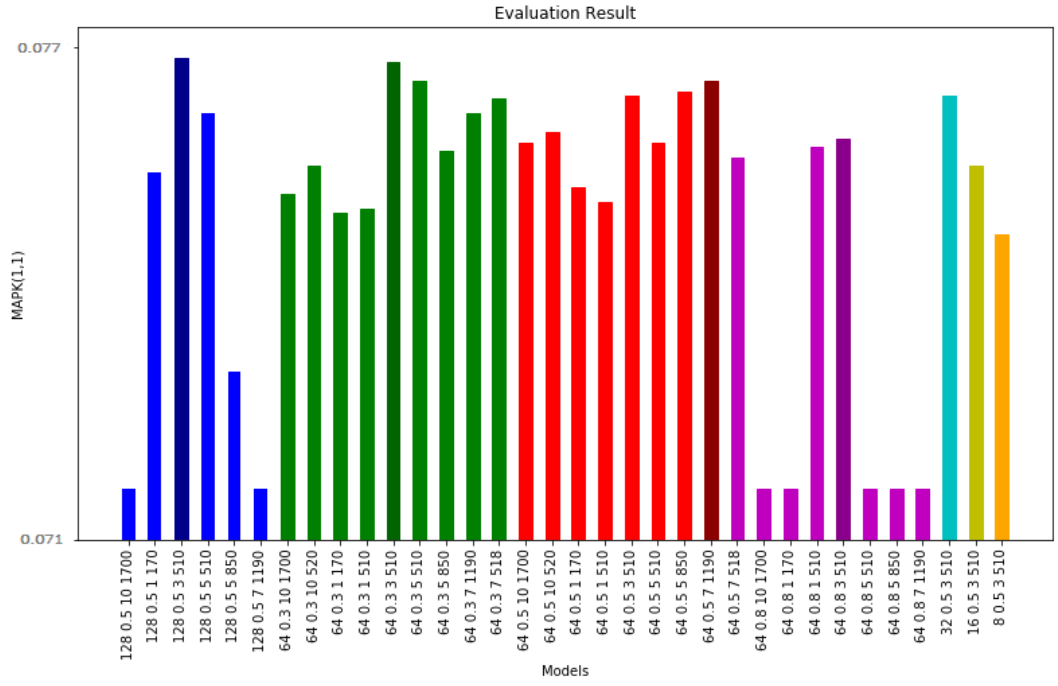


Figure 27 - Comparison of results for different model configurations.

embedding size	training time (mins.)
128	909
64	888
32	827
16	822
8	802

Table 8 - Training time per embedding size.

Regarding the execution time, as it is shown in *Table 8*, the total time required for training the model using an embedding size of 32 is 9% less than the time it takes to train the embedding of size 128. The cited table shows that using values lower than 32 for the embedding size, allows to improve performance by reducing the training time a bit more, but as we can see in *Figure 27* the quality of the results is not good enough. Discovering a good balance between quality and performance can be advantageous in order to process bigger datasets.
Considering an embedding size of 128, the training phase takes about 15 hours, which can be reduced in 21 minutes approximately If we train the model using an embedding of size 64. The server used for training is an Intel(R) Xeon(R) 4860@2.27GHz with 32 cores and 40 GiB RAM (without GPU). It's worth mentioning that the code used for training was not optimized for GPU usage. Using an optimized code with GPU support probably can reduce these times significantly, although is something that was not explicitly tested.

The output of the procedure is a vector representation of the top 40000 domains. Then, the cosine distance between these vectors is computed to get the nearest domain names for each one.
In what follows, the first results that were obtained by our first candidate model are presented.

Visual Inspection: semantic similarity

Before moving to a more formal analysis and comparison of this model against others, let's see some examples about semantic similarity by visual inspection to have a sense of whether the learned vectors are able to capture meaningful semantic information or not. In order to do this, we present the results obtained when asking the model for the nearest vectors to a given domain name, that is, the most similar sites to the given domain name.

Most similar domain names to *subrayado.com.uy* (tv news):

Domain name	***Type***	***Cosine distance***	***Observations***
subrayado.com	Non existent domain	0.839	Same domain, but without country code ‘uy’
tutv.com.uy	press, tv	0.831	Domain does not exist anymore (Jan-2019)
lr21.com.uy	press, newspaper	0.786	
eldiario.com.uy	press, newspaper	0.771	
diariolarepublica.net	press, newspaper	0.770	Alias for republica.com.uy
telenocheonline.com	press, tv news	0.767	Alias for telenoche.com.uy
informarte.com.uy	press, radio	0.765	
teledoce.com	press, tv news	0.756	
laprensa.com.uy	press, newspaper	0.736	
uruguayaldia.com.uy	news	0.733	Domain does not exist anymore (Jan-2019)
unoticias.com.uy	news	0.732	Domain does not exist anymore (Jan-2019)
uypress.net	press, newspaper	0.728	
diarioelpueblo.com.uy	press, newspaper	0.714	

Table 9 - Most similar domain names to subrayado.com.uy.

Most similar domain names to *autoblog.com.uy* (cars blog, cars reviews)

Domain name	***Type***	***Cosine distance***	***Observations***
area75.com.ar	Cars design	0.896	Argentine site
gonzalorodriguez.org	road safety, road traffic crashes	0.895	Nonprofit uruguayan organization
suzuki.com.uy	car brand in Uruguay	0.894	Suzuki brand in Uruguay
mundoautomotor.com.ar	cars blog, cars reviews	0.871	Argentine site
autos-chinos.com	cars blog, cars reviews	0.870	
peugeot.com.uy	car brand in Uruguay	0.856	Peugeot brand in Uruguay
fiat.com.uy	car brand in Uruguay	0.853	Fiat brand in Uruguay
autoanuario.com.uy	cars blog, cars reviews	0.832	
citroen.com.uy	car brand in Uruguay	0.829	Citroen brand in Uruguay
autoschinos.com.uy	cars blog, cars reviews	0.825	
cosasdeautos.com.ar	cars blog, cars reviews	0.816	Argentine site
masautos.com.uy	cars	0.811	Domain does not exist anymore (Jan-2019)
cochesyconcesionarios.com	Cars prices, price comparisons	0.811	
volkswagen.com.uy	car brand in Uruguay	0.80	Volkswagen brand in uruguay
cars-magazine.com.ar	cars blog, cars reviews	0.80	Argentine site

Table 10 - Most similar domain names to autoblog.com.uy.

Table 9 and *Table 10* give strong evidence about the model's capability for capturing semantic information about domain names. In the first one, when looking for similar sites to *subrayado.com.uy* (the website of a tv news from Uruguay) we can see that all similar domain names belong to websites related to press and news, from different types, written press, tv channel and even radio. An interesting observation here is that the procedure was able to learn embeddings for domain names that formally do not exist, like the case of *subrayado.com* suggesting that many queries to the DNS system were asking for the domain name without the country suffix. In some way, this is a common error, and it gives some insights about how people browse the web. Other interesting thing that is observed is that all the most similar sites to *subrayado.com.uy* are sites from Uruguay. Probably, we can think that when people want to check for news, it is very dependent on the location where people live. This is not as strict in the second example about cars where information about cars is more general and it is not as dependent to the location as in the news case.

The second example shows similar sites to *autoblog.com.uy* (a blog with news about cars and users reviews). We can see that all the similar domain names found by the model are very related, going from cars news, cars brands, cars design, even a site for road safety. As it was pointed out above, in this case, blogs and forums from the region are found by the model (not only from Uruguay), which make sense since general information about cars is the same. Anyway, we see again the same location dependent pattern when users look for brands and prices, in this case we can see that all the related sites are from Uruguay. This also make sense, since the source domain name (*autoblog.com.uy*) is an uruguayan website and most of its audience comes from uruguay, therefore after people check for user reviews about cars they move to some of the car brands in the Uruguayan market, probably to check about availability, prices, etc.

Visual Inspection: analogies through vector operations

In the original *Word2Vec* paper [56], it was mentioned that the linear structure of the *Skip-Gram* model allows analogical reasoning using simple vector operations. For example, the addition of vectors works as an *AND* logical function: the words near to the addition of two vectors will be words that are close to both original words. In our case, domains close to the

addition of the vectors that represent two particular domains will be the ones that are commonly accessed in conjunction with the added domains. *Table 11* shows some simple addition analogies between domains names. Also, in [56] it was shown that the hidden linear structure of the resulting vector space allows other more sophisticated operations, for example vec(*king) + vec(man) − vec(woman) ≃ vec(queen)*. This is particularly interesting for what is known as analogical reasoning. *Table 12* shows some more complex analogies using addition and subtraction.
In both tables, we show one of the 10 domains nearest to the resulting vector. Visual inspection verifies that our model can embed semantic relationship between domains and it allows applying analogical reasoning for understanding complex relationships .

v_1	v_2	$v_1 + v_2$
Ministry of tourism of Uruguay (turismo.gub.uy)	Ministry of tourism of Argentina (turismo.gov.ar)	Argentina's migration office (migraciones.gov.ar)
Ministry of tourism of Uruguay (turismo.gub.uy)	Ministry of tourism of Argentina (turismo.gov.ar)	Uruguayan travel site (pasaporteuruguay.com)
City government (montevideo.gub.uy)	Bus routes finder (montevideobus.com.uy)	City maps (mapred.com)
Airline (lan.com)	Hotel booking (booking.com)	Travel information (tripadvisor.com)

Table 11 - Logical analogies using addition.

v_1	v_2	v_3	$v_1 + v_2 - v_3$
City government (montevideo.gub.uy)	Bus terminal (trescruces.com.uy)	Other city government (rocha.gub.uy)	Bus line connecting cities (copsa.com.uy)
City government (montevideo.gub.uy)	Bus terminal (trescruces.com.uy)	Other city government (rocha.gub.uy)	Post site (correo.com.uy)
Shopping mall in city A (puntashopping.com.uy)	City B government (montevideo.gub.uy)	City A government (maldonado.gub.uy)	Bus terminal in city B (trescruces.com.uy)
Soccer site (tenfield.com)	Soccer club A fan page (campeondelsiglo.com)	Other soccer site (ovacion.com.uy)	Soccer club B fan page (lavozdenacional.com)

Table 12 - Logical analogies using addition and subtraction.

Performance metrics and the quality of the results obtained by this first model will be discussed deeper in Section 4.7 when the analysis and comparison of the differents approaches and generated models are analyzed. As a natural evolution of this model, following our intuition and also supported by some of the improvements presented in [65], we think that weighting words inside the context can help to improve the quality of the learned vectors. For this reason, in the next section we will explore a

variation of *Word2Vec*, using an approach similar to *App2Vec*[66] and giving higher weights to domain names that belong to the same context but were requested closer in time.

4.5. Adding time factor with App2Vec

In the previous section, a first model based on *Word2Vec* was shown as a good option for getting vectors representations that are able to capture meaningful semantic information about Internet domain names as well as perform analogical reasoning by applying simple vectors operations with the resulting embeddings.
In this section, a variation of *Word2Vec* applied to the DNS traces is presented. We call this variation an *App2Vec-like* model since it's inspired by the previous work in [66] which its authors called *App2Vec*. The core idea behind this is simple an intuitive, and it's based on applying different weights to domain names inside the context window of the *Word2Vec*'s *CBOW* architecture depending on how far they are from the middle domain name (the target domain name to be predicted when training the network).

If we recall from Section 3.2.2.3.2, in its standard behaviour, the *CBOW* architecture is similar to the standard bag of words, where order doesn't matter. It tries to predict the middle word in a sliding window by averaging the embeddings of the remaining words in the window, and all the words contribute equally to compute the final average. The improvement proposed in *App2Vec* suggests using a weighting factor that is applied to each word vector in the context window before the final average. By doing this, words in the context that appear close to the middle word receive a higher weight than those that are farther away, therefore contributing more to the resulting average vector. Later in [65], *Facebook* AI's researchers use a similar idea based on a position dependent weighting schema as one of their tricks to train high-quality word vector representations.

But although the core idea of weighting words in the context seems to be simple, the concepts of *far* and *close* is not trivial in our scenario. If we were using a standard text corpus for learning word embeddings as a classic *NLP* problem, then given any word in the context, we could use the

number of words that exist between this word and the target/middle word as a distance metric for calculating the weighting value.
But in the problem that we want to solve, it could be dangerous to apply the same concept of distance. In order to understand why, let's illustrate an example. Suppose we are using a context window of size 1 and we have $(d1, d2, d3)$ as the sequence (ordered) of domain names that represents the current sliding window during the *CBOW* model training. If we use the number of domain names after/before the target/middle word as metric for weighting context words then, domain names $d1$ and $d3$ will be weighted equally. But, what if domain name $d1$ was requested 3 minutes before $d2$ and $d3$ was requested 3 hours later? Knowing this new information, would you weight $d1$ and $d3$ equally? Which sites do you think are more related, $d1$ and $d2$ or $d2$ and $d3$?. These are some of the the questions that we asked ourselves when thinking about how to leverage the time information that we have available in the raw DNS queries and we did not use for our first model based on *Word2Vec*.

The reader probably noticed that in Section 4.2 we had defined a preprocessing step which its objective was to break long traces (with long time duration) into shorter ones (no more than 5 minutes). This trick, in some way was addressing the issue of long time intervals in the same context window. The usage of an arbitrary 5 minutes long window was a naive, yet effective strategy, that allowed us to train our first candidate model. Now, in this section our goal is to explore the usage of an *App2Vec-like* model that can deal with long time gaps between domain names in a same context window natively, and thus, removing the previously mentioned preprocessing step.
In order to do this, we need to include the time information regarding when each domain name was requested. More precisely, in order to train the *App2Vec-like* model, we rebuilt the corpus changing domain names traces from $(d_1\ d_2\ \dots\ d_n)$ to $(d_1\ g{:}x_1\ d_2\ g{:}x_2\ \dots\ d_n\ g{:}x_n\ d_{n+1})$ where d_i are valid domain names in the vocabulary and $g{:}x_i$ is used to indicate that the time gap between d_i and d_{i+1} is x_i. Time gaps x_i are float values that represent the difference in minutes between the DNS queries received in the DNS server for domain names d_i and d_{i+1} respectively. After doing this, we removed the preprocessing step that was breaking the user's trace into many sentences, and thus, sentences now are not limited to a maximum of 5 minutes duration. As we will see later in this section, we were able to

confirm that changing the maximum time difference allowed between domain names in a sentence does not affect the final results.

Running the experiments

A set of experiments were conducted to evaluate how a weighting schema based on time gaps could affect the quality of the learned vectors in our scenario. After contacting with the *App2Vec*'s authors we got access to the implementation they used in [66] where a slight modification to the original *CBOW* architecture is implemented to support a weighting schema based on time gaps. A detail here is that the core library used for the *App2Vec* implementation was the well known *Gensim* package for *Python* (widely used in *NLP* projects), hence we decided to move from *Tensorflow* to *Gensim* as the underlying technology for running the new experiments and for comparing the models' results. It's worth mentioning that the *Gensim* package provides a high level interface for the *Word2Vec* algorithms for both the *Skip-Gram* and the *CBOW* architectures, being great for easy and quick prototyping and experimentation. *Gensim* has been used in over a thousand research paper and student theses according to *Google Scholar*[54]. Furthermore, it uses *NumPy*[55], *SciPy*[56] and *Cython*[57] for high performance execution, being specifically designed to handle large text collections, using data streaming and efficient incremental algorithms. [99] describes the initial design decisions behind *Gensim* and the Ph.d. thesis in [25] shows details about the algorithmic scalability of distributional semantics that are implemented in this *Python* package.

In regards to the evaluation method, we continue using the *MAP@K* metric defined in Section 4.3, giving importance to the order in the predictions. More precisely, models were evaluated using the following (k_{actual} , k) pairs: *(1, 1), (5, 5), (10, 3), (10, 5)* and *(10, 10)*. Having said that, in what follow, the *MAP@K* metrics obtained using different configurations of the *App2Vec-like* model are presented. We experiment using different time factors for the weighting schema, different sizes for sentences and context window. We also evaluate the convergence time by measuring the

[54] https://scholar.google.com/
[55] http://www.numpy.org/
[56] https://www.scipy.org/
[57] https://cython.org/

performance of the model during the training phase for different number of epochs[58].

Experiment 1: evaluating time factor (default configuration)

Since the *App2Vec-like* model applies a weighting schema to the *CBOW* architecture of *Word2Vec*, in the first round of experiments we decided to compare the *App2Vec-like* model vs the standard *CBOW* architecture. By doing this, we can effectively measure the change in performance that is explained exclusively by the weighting schema based on time gaps. Later, in Section 4.7, the performance of this *App2Vec-like* model is analyzed and compared considering the other candidates models that were evaluated in this research (the *Skip-Gram* architecture of *Word2Vec* used by our first candidate model and a *FastText* based model that will be presented in Section 4.6).
For the time factor value required by the *App2Vec-like* model, the default value of *0.8* is used (the same value suggested in [66]). In regards to the window size parameter (context length), it is not specified in the *App2Vec*'s paper, but we found that *5* is the value used in the source code of the *App2Vec*'s implementation, hence it is the value that we use for our initial experiments. Other parameters are kept with the same configuration used by our first candidate model (see Section 4.4), and although the *App2Vec-like* model should work well without any restriction about the size of the sentences, for the initial experiments we decided to keep using the same restriction regarding the maximum difference in time allowed between domain names in a same sentence (5 minutes maximum). Later, other experiments will show that this restriction does not affect the results of the *App2Vec-like* model and it can be safely removed.

Figure 28 shows the *MAP@k* metrics that we obtained for both models (*App2Vec-like model using weighted version of the CBOW architecture based on time gaps vs the standard CBOW architecture of Word2Vec*) for different values of k_{actual} and k. Training ran for 21 epochs, saving partial results every 3 epochs. As we can see in *Figure 29*, the stabilization points (for both models) were found around the 6th and 9th epoch. From the 9th to the 21st epoch, the metrics reported by both models were practically identical, only some insignificant difference was noticed

[58] One epoch means one pass of the full training set through the neural network

probably due to the random behaviour associated with the negative sampling method.

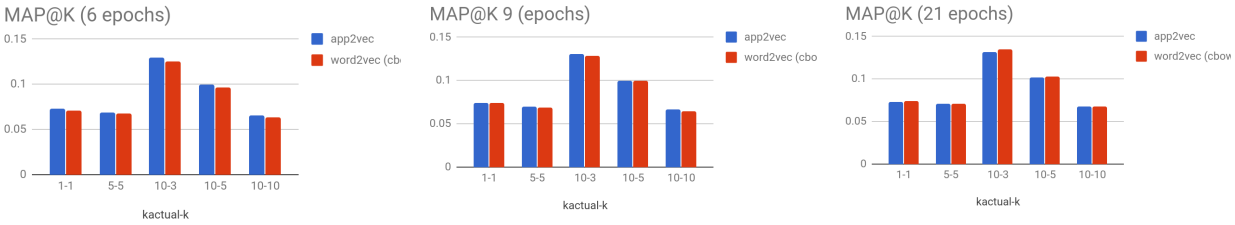


Figure 28 - App2Vec-like model (weighted cbow) vs Word2Vec (standard cbow).

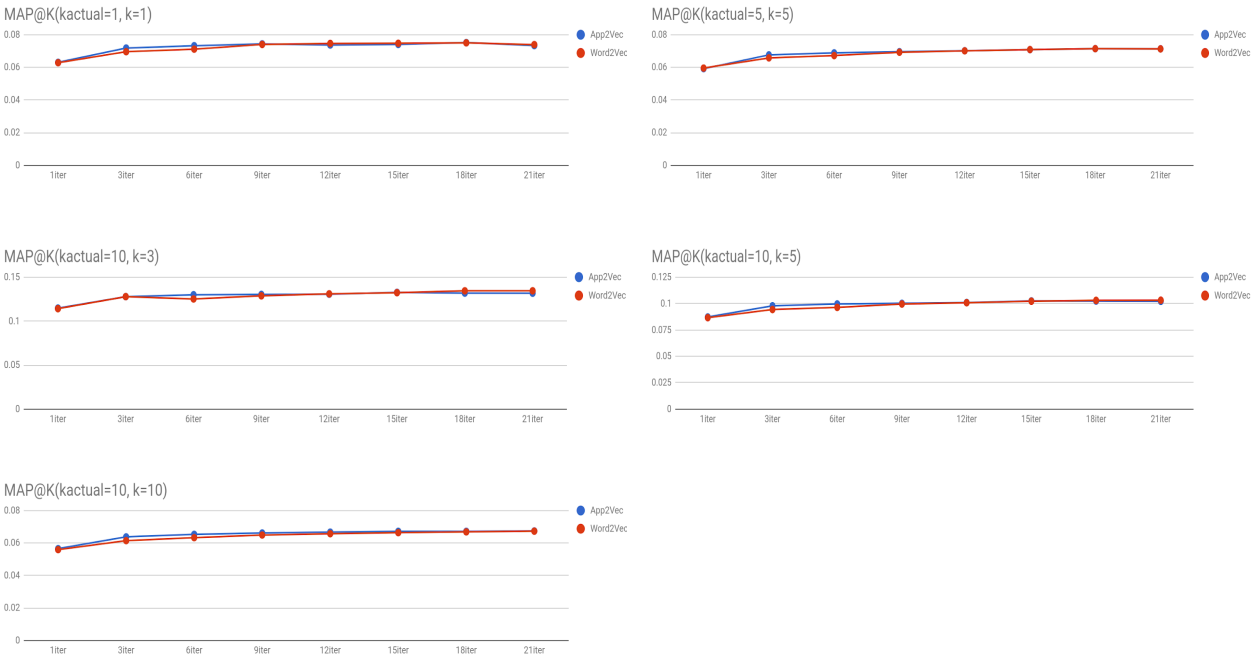


Figure 29 - Evolution of MAP@k during the training phase.

As we can see in *Figure 30* the best results (for both models, in all epochs) are obtained with k_{actual}*=10* and *k=3*, that is, when looking for the top 3 predictions of ours models into the results we get from the *Alexa*'s service for similar sites (taking in consideration the order in our models' predictions).

But unfortunately, from these results we cannot say that the weighted schema used by the *App2Vec-like* model performed better than the standard *CBOW* architecture of *Word2Vec* as we expected. The *App2Vec-like* model only outperformed the standard *CBOW* during the firsts few epochs (until the 9th epoch), but without being a significant difference. This could give some evidence that the *App2Vec-like* model is able to learn the embeddings a bit faster than the standard *CBOW*, but when working with a very big dataset like in our case, once both models arrive to the stabilization point, they achieve similar results.

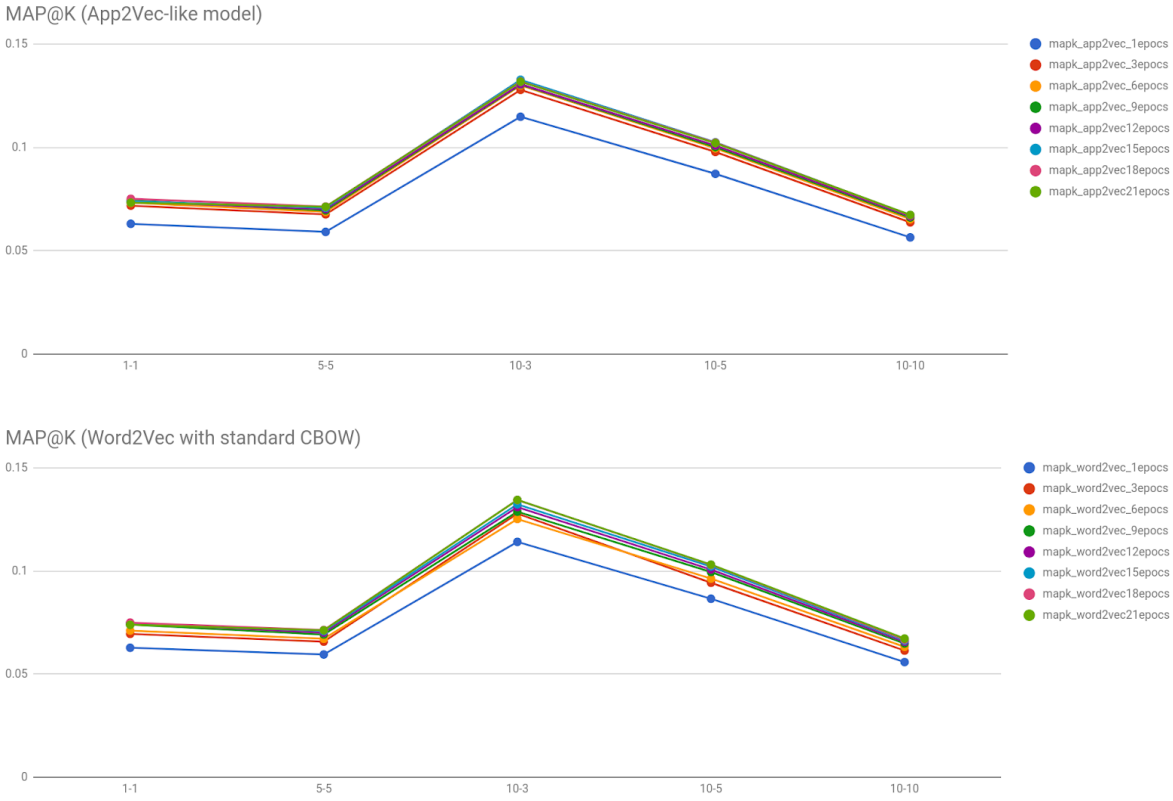


Figure 30 - MAP@k metrics for different configurations of k_{actual} and k measured at different epochs.

In the next experiments we will try to figure out if we can find a better configuration of hyperparameters for the *App2Vec-like* model that can get better results than the ones obtained during the first round of experiments just presented.

Experiment 2: tuning the time factor value

The second round of experiments have as main objective to find the best value used by the *App2Vec-like* model to weight domain names inside the context window based on time gaps.
In the initial experiments the default value of *0.8* (original value suggested in [66]) had been used. Now, we repeat the experiments varying the value of the time factor, taking a value from the set: *{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9}* in each experiment. Value of *0* is not considered because the final average would be always zero, and similarly, value of *1* is excluded because it corresponds to the special case of the standard *CBOW* where all words in the context contribute equally to the average embedding.

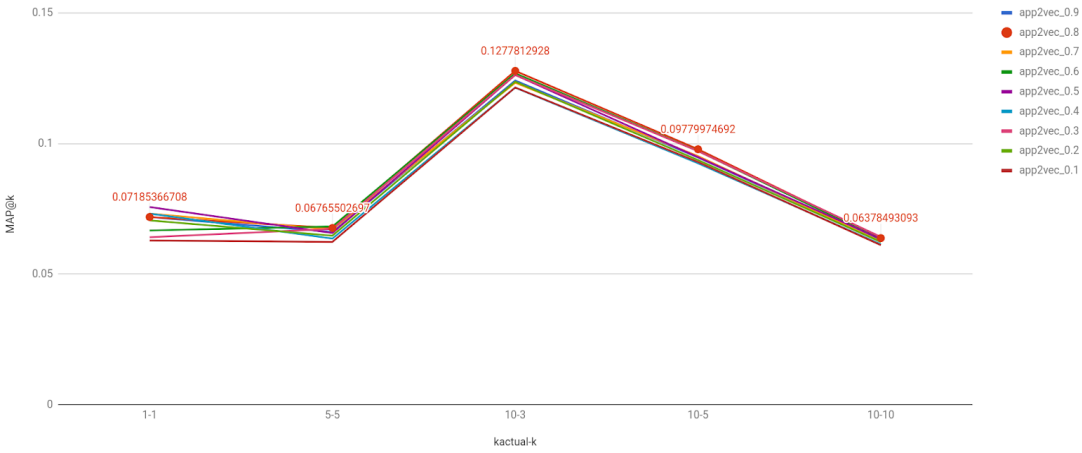


Figure 31 - MAP@k metrics for different values of the App2Vec's time factor.

Figure 31 shows the different *MAP@k* metrics that were obtained when changing the time factor value of the *App2Vec-like* model. These metrics were obtained evaluating each model configuration immediately after having completed the third epoch of training. As we can see, the default value of *0.8* (the same value that we used during the initial experiments) was the one that achieved the best results if we take in consideration the different values of k_{actual} and k. For this reason, for the final experiments we will continue using the default value of *0.8* for the time factor value configuration.

Experiment 3: evaluating sentence length (based on traces duration)

In the previous experiments we were still using the restriction of a limited sentence size based on the maximum difference in time allowed for domain names in the sentence (5 minutes maximum). Now, in the next experiments we want to validate our hypothesis about the expected behaviour of the *App2Vec-like* model and we want to confirm that the sentence size configuration does not affect the results since the important dependencies between domain names are now defined by the time factor and the weighting schema.
We evaluate the *App2Vec-like* model repeating the experiments by training the models during 6 epochs and evaluating the *MAP@k* metric for different values of k_{actual} and *k* for different sentence sizes: 5, 10 and 30 minutes.

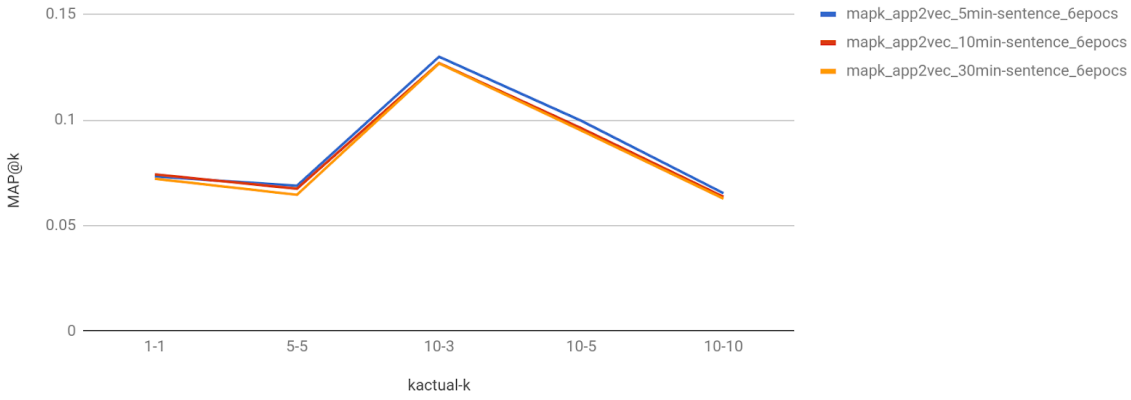


Figure 32 - MAP@k metrics for different sentence sizes.

As we can see in *Figure 32*, after completing the 6th epoch the results are practically identical. We do not observe any significant difference in the *MAP@k* metrics, and thus, validating the idea that this value is not relevant any more when using a weighting schema. For this reason for the final round of experiments we remove any restriction related to the sentence size.

Experiment 4: evaluating window size

The final experiments try to measure the effect of the window size. We repeat the experiments for different window sizes: *3* (same value used

by our first candidate model), *5* (same value used in the original *App2Vec's* implementation) and 10 (a new value that duplicates the default window size used in *App2Vec*).
Figure 33 shows the results that we got for these experiments. We can see that values of *5* and *10* for the window size get almost identical results (the only case that shows a difference in favor of a window size of 5 is when k_{actual}*=1* and *K=1*).

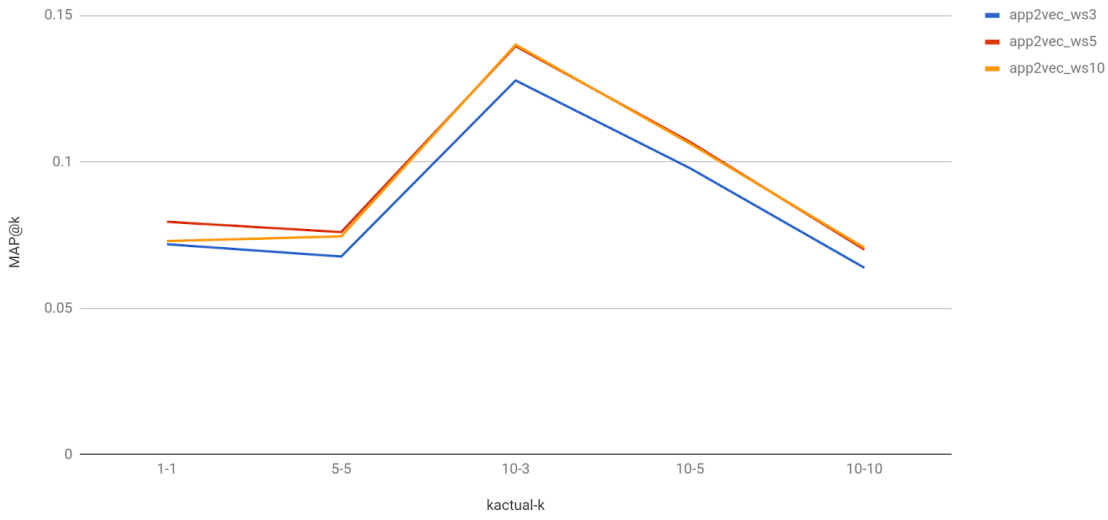


Figure 33 - MAP@k metrics for different window sizes.

The very pretty similar results for window sizes of *5* and *10* was something that we expected in some way, because similarly to what happened with the sentence length, the distance between domain names are now better governed by the time gaps between domain names instead of a fixed count of instances that occur right after or before a target domain name.
According to these results, a window size of *3* seems to be a bit restrictive for our particular scenario and data, leaving domain names out of the context that could affect favorably if they were included. A window size of *5* (or greater) give the optimal vectors. Once results are the same (like when using a window size of *5* or *10*) greater values for this hyperparameter do not have any different effect because the new domain names that are included in the window are too much distant in time to the target domain name, therefore their weighted value affect the average embedding

insignificantly. For this reason, we conclude that a size of *5* for the context window (the same default value that we have been using from the initial experiments with the *App2Vec-like* model) is the best option to use.
Having said that, we can take the comparisons presented in the first round of experiments as a good reference to compare the *App2Vec-like* model vs the standard *CBOW* architecture of *Word2Vec*. This comparison had already shown that in our scenario and with our specific data, the difference between these two approaches is not so big, and results are pretty similar (only an insignificant difference in favor of the weighted schema is observed in general). In Section 4.7 more details about this *App2Vec-like* model and a general comparison with the other candidate models are presented. But before that, let's see by visual inspection some of the similarities that are found by this second candidate model.

Visual Inspection

Before ending this section, and similarly to what we did with our first candidate model some results obtained by this new model are shown. By doing this, we can get some sense of the model's quality directly from observation rather than trying to understand whether the evaluation metrics are good or not.
The hyperparameter configuration of the final *App2Vec-like* model use the best configuration found after running the set of experiments already explained before, that is:

- Architecture: weighted *CBOW*
- Distance factor (for the weighting schema based on time gaps): 0.8
- Window Size: 5
- Sentence length: unlimited
- Vector size: 64

Most similar domain names to *subrayado.com.uy* (tv news):

Domain name	***Type***	***Cosine distance***	***Observations***
tutv.com.uy	press, tv	0.771	Domain does not exist anymore (Jan-2019)
teledoce.com	press, tv news	0.763	
subrayado.com	Non existent domain	0.752	Same domain, but without country code ‘uy’
diariolarepublica.net	press, newspaper	0.694	Alias for republica.com.uy
lr21.com.uy	press, newspaper	0.688	
eldiario.com.uy	press, newspaper	0.683	
cienporcientopapal.com	Uruguayan soccer club news	0.647	Domain does not exist anymore (Jan-2019)
diariouruguay .com.uy	press, newspaper	0.638	
vamosuruguay.com.uy	Politics	0.635	
uypress.net	press, newspaper	0.634	
www.elnaveghable.cl	press, newspaper	0.629	Chilean site
elbocon.com.uy	press, newspaper	0.621	

Table 13 - Most similar domain names to subrayado.com.uy (using App2Vec-like model).

Most similar domain names to *autoblog.com.uy* (cars blog, cars reviews)

Domain name	***Type***	***Cosine distance***	***Observations***
gonzalorodriguez.org	road safety, road traffic crashes	0.809	Nonprofit uruguayan organization
area75.com.ar	cars design	0.805	Argentine site
suzuki.com.uy	car brand in Uruguay	0.797	Suzuki brand in Uruguay
autos-chinos.com	cars blog, cars reviews	0.785	
ayaxonline.com	car dealership	0.773	
autoanuario.com.uy	cars blog, cars reviews	0.766	
cochesyconcesionarios.com	cars prices, price comparisons	0.734	
greatwall.com.uy	car brand in Uruguay	0.729	Greatwall brand in Uruguay
vladimir.com.uy	car dealership	0.725	
imgci.com	-	0.718	Domain does not exist anymore (Jan-2019)
fiat.com.uy	car brand in Uruguay	0.717	Fiat brand in Uruguay
www.autosonline.cl	cars blog, cars reviews	0.710	Chilean site

Table 14 - Most similar domain names to autoblog.com.uy (using App2Vec-like model).

Table 13 and *Table 14* give strong evidence about the model's capability for capturing semantic information about domain names. Visual inspection of these results also confirms that the performance of this *App2Vec-like* model is comparable with our first candidate model. We can

see that the results for the most similar sites to *subrayado.com.uy* and *autoblog.com.uy* are pretty much the same than the results we got with our first candidate model using *Word2Vec*. Only some changes in the order of the results, and some new sites are added or removed from the list of similars, but keeping in general the same semantic meaning for the similar sites in both cases.

4.6. Considering sub-word level with FastText

A common property shared by both of our previous candidate models (as well as by all classic word embedding techniques) is that words are treated as indivisible tokens, as single units. But in our scenario, words correspond to domain names, and the composition and the morphological structure of domain names carry important information about the meaning of the domain. For example, if we have a website called *futbol.com* and another website called *futbolparatodos.com* then it's high probable that these two websites have some kind of content related to the subject of futbol, therefore being semantically related. The reader should notice, that we are saying *"high probable"*, and that's because we are formulating an hypothesis based just on the semantic information provided in the string that represents the site's name but we do not have access to the content hosted in each site to validate that hypothesis.

Hence, the morphological information about domain names could be crucial information that we have available but we have not used yet in our problem.
For this reason, in this section a set of experiments that take into account the morphological structure of domain names are carried out with the objective of enhancing our algorithms and models for finding the best vector representation of Internet domain names.
In order to do this, we experiment with *FastText*[59] which extends the functionality of *Word2Vec* to work with sub-words and thus, adding important information to the embeddings that is related to the morphological composition of the words.

[59] https://fasttext.cc/

As it was explained in Section 3.2.2.5, formally talking, *FastText* is not a method, algorithm or technique by itself. *FastText* is an open-source, free, lightweight library that allows users to learn text representations and text classifiers [70]. It can be used either in unsupervised mode to learn word embeddings [71] or in a supervised mode using a labeled dataset (one label and sentence per line) to train a text classifier [72]. For a practical use case of *FastText* in supervised mode, the reader can see the github page that we have created in [100] where *FastText* is used for training a text classifier. For the experiments presented in this section, the unsupervised mode of *FastText* has been used to learn vector representations of domain names by learning character *n-grams* embeddings and aggregating them to compute the final words embeddings.

We used the *FastText Python wrapper*[60] (included in the *Gensim* package) that provides direct access to the original and optimized implementation of *FastText* in *C*. This is noticeably faster than the pure *Python* implementation available in *Gensim*.
Since *FastText* is an extension of *Word2Vec*, it can be configured to work with the same two architectures supported by *Word2Vec*: the *Skip-Gram* and the *CBOW* architectures. When using the *CBOW* architecture, the current word in the context window is predicted from its context words. On the other hand, when using the *Skip-Gram* architecture the context words are predicted from the current word. Once the training is completed, the final weights associated to the hidden layer are used as the embeddings for the words. In this work we used the *Skip-Gram* architecture since it has proven to perform better with large datasets [36]. The reader can see Section 3.2.2.3.1 and 3.2.2.3.2 to review the details about these two architectures originally proposed in the first *Word2Vec*'s paper in [56].

The most critical hyper-parameters that we needed to tune are: *minn* (minimum *n-gram* size) and *maxn* (maximum *n-gram* size). Sub-words are all the substrings contained in a word with size between *minn* and *maxn*. Also, the word itself is considered to be in the set of its *n-grams* [71]. If *maxn* is set to 0, or lesser than *minn*, no character *n-grams* are used, and the model is effectively reduced to *Word2Vec* [74]. *Figure 34* shows the results that we obtained when computing the *MAP@k* metric

[60] https://radimrehurek.com/gensim/models/wrappers/fasttext.html

(k_{actual} = *3* and *k = 1*) for different combinations of *minn* and *maxn* (for this experiments we continue using an embedding size of 64 dimensions).

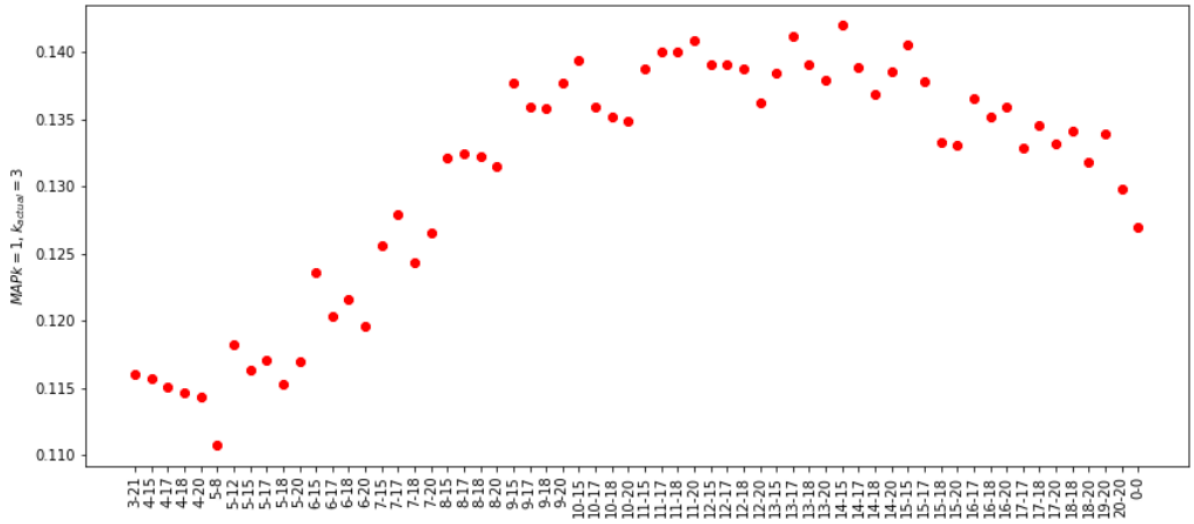


Figure 34 - Comparison of n-gram size (for k = 1 and k_{actual} = 3).

The experiments were repeated for different combinations of *(k, k_{actual})* pairs and *n-gram* lengths between *(0, 0)* and *(20, 20)*. For each experiment 6 epochs of training were executed and then (once the training was stable) the *MAP@k* metric was evaluated for each configuration of k_{actual} and *k*. The special case of *minn = 0* and *maxn = 0* corresponds to the test case when no *n-gams* are used. For that special test case we were able to confirm that the results are exactly the same as the results that we got when using the *Word2Vec* version with *Skip-Gram*.
In order to find the best configuration for the *minn* and *maxn* hyperparameters, we used two different evaluation criterias that are described below.

Evaluation criteria 1:

The first evaluation criteria consisted in counting the number of times that each n-gram range (an *n-gram* range is defined by a pairs of *minn* and *maxn* values) performed the best.
Hence, given all the *25* possible combinations of *K* and k_{actual} from the set *{1, 2, 3, 5, 10}* we measure the MAP@k for each combination and we add

1 for the *n-gram* range that achieves the best performance (the *n-gram* range that achieves the highest *MAP@k)*. *Table 15* shows a summary with the number of times that each *n-gram* range performed the best. *N-gram* ranges that never achieved the highest *MAP@k* are not included in this table.

minn	*maxn*	*Best (number of times)*
11	17	5
10	15	4
12	17	3
14	15	3
15	15	3
13	15	2
14	17	2
15	17	2
11	20	1

Table 15 - Number of times that each n-gram range performed the best.

Evaluation criteria 2:

In the first evaluation we used a hard criteria where in each of the 25 evaluations of the MAP@k metric each n-gram range either added 1 or 0 depending whether the n-gram range achieved the best result or not.
For the second evaluation criteria, we followed a similar approach but using a softer criteria. Instead of adding a value of 1 or 0, for each n-gram range we sum the results of the MAP@k metric for each of the 25 combinations of kactual and k values. In the end, the n-gram range that accumulates the highest value is the best option. *Table 16* shows the top 20 n-gram ranges order by the accumulated MAP@k value (descending).

minn	*maxn*	*sum of MAP@k*
11	17	3.787
10	15	3.782
15	15	3.781
14	15	3.779
12	17	3.774
11	18	3.770
11	20	3.766
13	17	3.765
14	17	3.757
11	15	3.755
12	15	3.742
13	15	3.742
14	20	3.739
14	18	3.734
9	20	3.734
10	20	3.733
15	17	3.732
12	18	3.731
16	17	3.730
13	18	3.725

Table 16 - Top 20 n-gram ranges ordered by sum of MAP@k

Supported by these two evaluation criterias we found that the optimal configuration is using *minn = 11* and *maxn = 17,* therefore it is the configuration that we will use for our *FastText* based model. Other configurations like *minn = 10* and *maxn = 15* seems to be good too. Furthermore, we can see in *Figure 34* that using *n-grams* with a number of characters greater than *10* outperforms the results achieved by the

Word2Vec with *Skip-Gram* model (configuration with *minn = 0*, *maxn = 0*). But since we need to choose one, *n-gram* range between *11* and *17* is the winner.

It is important to note that usually when working with standard text corpus the best results are obtained with much shorter *n-grams* of about 3 to 6 characters. In our case this very short *n-grams* are not very useful because they encompass words like *.www*, *.com or .com.uy* (among others) that do not help to determine the compatibility between different domains. By using longer *n-grams* we are able to include longer substrings of the domain names that are more adequate to determine whether two domains are similars or not.

Visual Inspection

Before ending this section, and similarly to what we did with our previous candidate models, some results obtained by this new model (*FastText* using the *Skip-Gram* architecture with an embedding size of 64 dimensions and *n-grams* length between 11 and 17) are shown. By doing this, we can get some sense of the model's quality directly from observation rather than trying to understand whether the evaluation metrics are good or not.

Most similar domain names to *subrayado.com.uy* (tv news):

Domain name	***Type***	***Cosine distance***	***Observations***
subrayado.com	Non existent domain	0.916	Same domain, but without country code ‘uy’
diariolarepublica.net	press, newspaper	0.836	Alias for republica.com.uy
eldiario.com.uy	press, newspaper	0.807	
lr21.com.uy	press, newspaper	0.799	
teledoce.com	press, tv news	0.792	
elecodigital.com.uy	press, newspaper	0.774	
causaabierta.com.uy	-	0.77	Domain does not exist anymore (Jan-2019)
unoticias.com.uy	press, newspaper	0.766	
radiouruguay.com.uy	press, radio, newspaper	0.766	
uypress.net	press, newspaper	0.742	
sangregoriodepolancodigital.com.uy	press, newspaper	0.73	Domain does not exist anymore (Jan-2019)
vivomontevideo.com	-	0.71	Domain does not exist anymore (Jan-2019)

Table 17 - Most similar domain names to subrayado.com.uy (using FastText model).

Most similar domain names to *autoblog.com.uy* (cars blog, cars reviews)

Domain name	***Type***	***Cosine distance***	***Observations***
gonzalorodriguez.org	road safety, road traffic crashes	0.854	Nonprofit uruguayan organization
autoanuario.com.uy	cars blog, cars reviews	0.845	
mundoautomotor.com.ar	cars blog, cars reviews	0.822	Argentine site
cochesyconcesionarios.com	cars prices, price comparisons	0.819	
area75.com.ar	cars design	0.806	Argentine site
autos-chinos.com	cars blog, cars reviews	0.803	
suzuki.com.uy	car brand in Uruguay	0.781	Suzuki brand in Uruguay
peugeot.com.uy	car brand in Uruguay	0.778	Peugeot brand in Uruguay
gonzaloruiz.com.uy	car dealership	0.774	
masautos.com.uy	cars	0.774	Domain does not exist anymore (Jan-2019)
autosenuruguay.com	cars	0.768	Domain does not exist anymore (Jan-2019)
mundoautomotor.com	cars blog, cars reviews	0.767	
rcristofano.com	car dealership	0.759	
autoschinos.com.uy	cars blog, cars reviews	0.753	
chana.com.uy	car brand in Uruguay	0.753	Chana brand in Uruguay

Table 18 - Most similar domain names to autoblog.com.uy (using FastText model).

Once again, we can confirm through visual inspection (from *Table 17* and *Table 18*), that the similar sites found by this new candidate model keep important semantic relationships among domain names. But since all our candidate models have shown good results through visual inspection and it's difficult to choose the best model just by a subjective feeling about which domains are more related, we need to compare them more formally.
For that reason, in the next section more details about this *FastText* model and a general comparison with the other candidate models are presented. We will see that this *FastText* model allows to perform vector operations like addition and subtraction for logical analogies and that it outperforms considerably the results obtained by the other candidate models (not only considering the *MAP@k* metric, also with other metrics like *recall*, *precision* and *F1-score*), hence being our best model for building the DNS vector space model. Furthermore, we will highlight an interesting property of this model that allows us to find vector representations for words that were not originally part of the training (a.k.a *out-of-vocabulary* or by its acronym *OOV*) which could be helpful in many applications.

4.7. Analyzing the results

In previous sections we have shown how to get vector representations of domain names (DNS embeddings) through the usage of different word embeddings techniques. In particular we showed that predictive models for learning word embeddings such as *Word2Vec*, *App2Vec* or *FastText* are suitable for the task of computing DNS embeddings by working directly with the DNS traces that result from a set of steps that preprocess the raw DNS log files (see Section 4.2 for a review of the preprocessing phase).
We have seen (through visual inspection) that the best configuration that we found for each candidate model is able to capture meaningful semantic information about domain names, and the operation of finding the most similar sites to a given domain name (by computing the cosine similarity between domain names vectors) gives effectively very good results (domain names in the results are sites in the same business activity or category).
Now, in this section we will present a more formal comparison of our candidates models by evaluating different metrics such as *MAP@k*, *recall*,

precision and *F1-score* (see Section 4.3) for a review of the evaluation framework) and analyzing their characteristics.

As explained in Section 4.3, we use the service called *find similar sites*[61] offered by *Alexa*[62] in order to retrieve the ordered top 100 most similar sites for a specific input domain name. We have this information for 14490 domains of our vocabulary (of 40000 domains).
During the training phase, our evaluation procedure uses this external, but well trusted information from *Alexa*, to give some measure of quality. To be more accurate, for every epoch of optimization and for all the 14490 domains that we were able to get information from the *Alexa*'s service, we compare the similarities between the actual *Alexa*'s response and the predicted similarities found by each of our candidate models. We evaluated the performance of each candidate model varying the number of predictions that are taken in consideration from our model (specified by the *k* value) as well as from the *Alexa*'s service (specified by the k_{actual} value). In each case we calculated the average *precision*, *recall* and *F1-score* over the prediction of all the 14490 domains under evaluation. In regards to the *MAP@k* metric, it was calculated using the named values for *k* and k_{actual}.

Given the 21 training epochs that were executed and the 25 possible combinations of k_{actual} and *k* values taken from the set *{1, 2, 3, 5, 10}* that were evaluated in each epoch, we have a total of *21 x 25 = 525* evaluation instances. *Table 19* shows a matrix M where $M(i,j)$ represents the number of times that model *i* outperformed model *j* using the *MAP@k* metric.
Table 19 shows clearly that most of the time *FastText* performed better than the other candidate models. Also, it's interesting to see that in 15 evaluation instances *App2Vec* outperformed *FastText*, this really took our attention and looking closer in those cases we were able to identify that those evaluation instances were all in the first epoch. Similarly, we had already noticed in Section 4.5 that the *App2Vec-like* model had achieved better results than the standard *CBOW* architecture of *Word2Vec* during the first epochs of training. This last observation is confirmed in *Figure 35* where we can see that *App2Vec* (red serie) outperforms *Word2Vec* with

[61] https://www.alexa.com/find-similar-sites
[62] https://www.alexa.com/

CBOW (blue serie) until the *16th* epoch. We notice that subsequents results are pretty similar for *App2Vec* and *Word2Vec* with *CBOW*.
Figure 35 is very representative of the comparison between candidate models for different values of k_{actual} and k during training. In order to make the analysis more reader-friendly we do not include all the charts comparing the different models for every value of k_{actual} and k in this section, but the reader can see them in Appendix B.

i \ j	*App2Vec*	*Word2Vec (cbow)*	*Word2Vec (skip-gram)*	*FastText*
App2Vec	0	347	129	15
Word2Vec (cbow)	176	0	79	18
Word2Vec (skip-gram)	396	446	0	16
FastText	**510**	**507**	**509**	**0**

Table 19 - Number of times a model performed better than other.

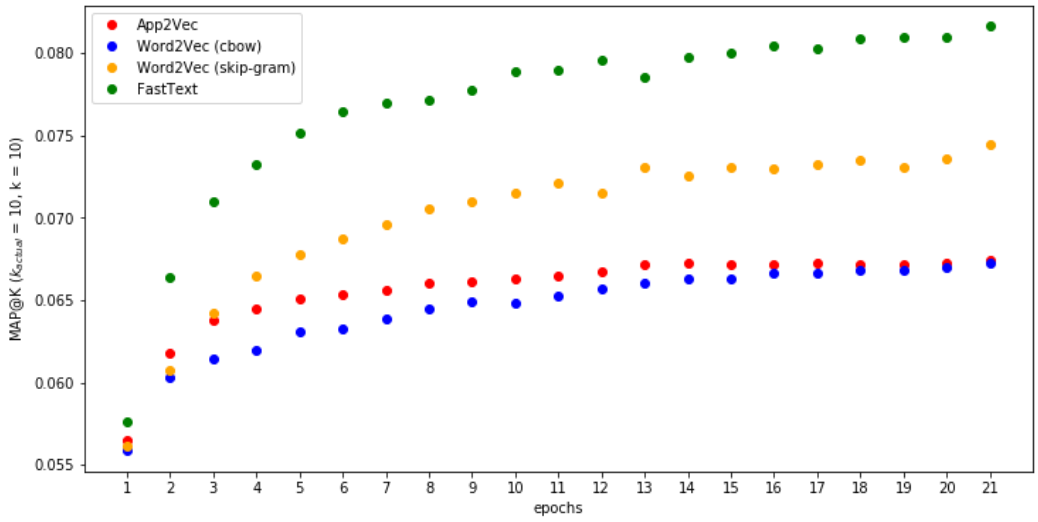


Figure 35 - MAP@k evolution for the different candidate models.

Other observations that we can highlight from *Figure 35* is that the *Word2Vec* with *Skip-Gram* version of our first candidate model that uses sentences with 5 minutes length (maximum) performs better than the *Word2Vec* version with *CBOW* and also than the *App2Vec-like* model, but without being as good as the *FastText* model. As we will see during this section, this pattern is a common factor in all the evaluations and comparisons that we executed.
For example, the order of models according to how well they perform regarding to the *MAP@k* metric is even more evident in *Table 20* that shows a matrix M where $M(i,j)$ represents the number of times that model *i* outperformed model *j* after the full training is completed. Once again, we can see that *FastText* is the best, followed by *Word2Vec* with *Skip-Gram* and we notice a slight superiority of *App2Vec* over *Word2Vec* with *CBOW*.

i \ j	*App2Vec*	*Word2Vec (cbow)*	*Word2Vec (skip-gram)*	*FastText*
App2Vec	0	16	0	0
Word2Vec (cbow)	9	0	0	0
Word2Vec (skip-gram)	25	25	0	0
FastText	**25**	**25**	**25**	**0**

Table 20 - Number of times a model performed better than other after training is completed.

The *MAP@k* metric has been also calculated for our baseline algorithms. The results are about *0.00005* for the *random guessing* algorithm and *0.016246* for the *popularity based* algorithm (see section 4.3.1 for details). All our candidate models and in particular the *FastText* based model outperforms considerably the results obtained by the baseline models.

After repeating the evaluation of the *MAP@k* metric for different values of k_{actual} and *k* we noticed that the *FastText* model obtained the best results in every tested scenario (for every combination of *k* and k_{actual}). The

highest *MAP@k* value obtained was 0.238 (with *k* = 1 and *kactual* = 10). After averaging the results, we conclude that the *FastText* based model is 10.5%, 17.8%, 17.8% and 435.5% superior than *Word2Vec* with *Skip-Gram*, *App2Vec, Word2Vec* with *CBOW* and the best baseline model (*popularity based*) respectively.

Now, in order to evaluate how well our candidate models perform using other evaluation metric, in *Figure 36* we show the results that we obtained using the *F1-metric*. The highest *F1-score* was 0.144 (with *k* = 10 and k_{actual} = 10). As explained in Section 4.3 the *F1-score* computes the harmonic mean between *precision* and *recall*, therefore when evaluating *F1-score* in some way we are evaluating both *precision* and *recall* at the same time, being the *F1-score* better when *precision* and *recall* are higher. Since most of the times we can increase recall by decreasing precision (and vice versa), *F1-score* is a common way for finding a good balance between both metrics. The reader can see Appendix A for an individual comparison of the precision and recall metrics as well as the results obtained using other configurations of the k_{actual} and *k* values.

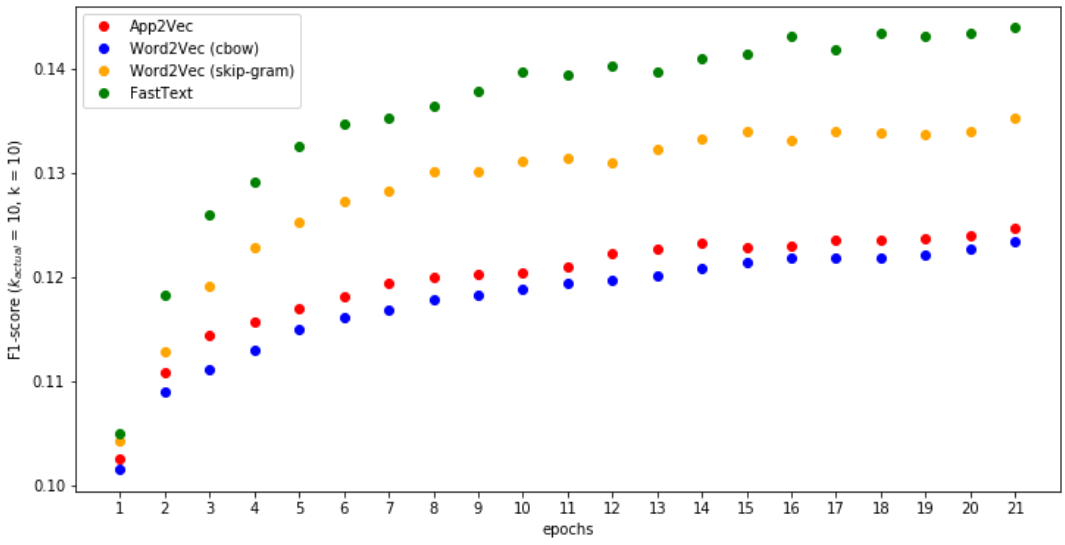


Figure 36 - F1-score evolution for the different candidate models.

As we can see, the trends shown in *Figure 35* and *Figure 36* are pretty much the same. Getting similar results for the relative comparison of the candidate models for different metrics gives strong evidence about the quality of the models.

A last comparison in *Figure 37* summarizes the different metrics that were evaluated once the training phase was completed. This last comparison is specific for $k = 5$ and $k_{actual} = 5$ but the relative difference between models for the different metrics are very similar for other values of k_{actual} and k and they can be seen in Appendix A.

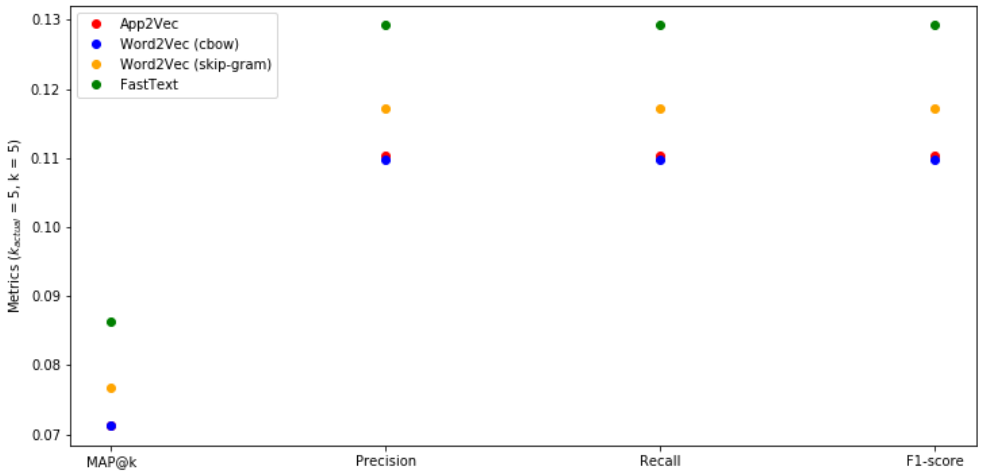


Figure 37 - Final metrics after training is completed.

Supported by these evaluations and comparisons, in this point we are confident about having found a good model based on *FastText* (with *Skip-Gram* architecture and *n-gram* range between 11 and 17 for sub-words) for finding semantically related domain names. But how good is this model regarding logical analogies? Unfortunately, we left out of the scope from this research the generation of an evaluation set for this task (it would be something interesting to address as future work). Anyway, similarly to what we did with our first candidate model based on *Word2Vec* with *Skip-Gram,* we show now (*Table 21*) some examples that give evidence that linear relationships between vectors can be leveraged by our *FastText* based model to perform analogical reasoning.

$v1$	$v2$	$v3$	$v1 + v2 - v3$
atlantida.com.uy *(site related to Atlantida, the main resort in Canelones city)*	***maldonado.gub.uy*** *(site for the Maldonado city government)*	***canelones.gub.uy*** *(site for the Canelones city government)*	***puntaweb.com*** ***puntadeleste.com*** *(sites related to Punta del Este, the main resort in Maldonado city)*
puntashopping.com.uy *(site for a shopping center in Maldonado city)*	***montevideo.gub.uy*** *(site for the Montevideo city government)*	***maldonado.gub.uy*** *(site for the Maldonado city government)*	***tiendasmontevideo.com*** ***montevideoshopping.com.uy*** *(sites for shopping centers in Montevideo city)*

Table 21 - Analogical reasoning using our FastText based model

Both examples presented in *Table 21* show 2 of the 3 domain names nearest to the resulting vector $v1 + v2 - v3$. Visual inspection suggests that our model could be used for analogical reasoning and thus being helpful for understanding complex relationships between domain names. Anyway, and although this examples look promising, more work needs to be done in this direction to validate through a more formal evaluation that our model is effectively good enough for the task of analogical reasoning.

Another interesting property of this *FastText* based model is regarding to domain names that do not exist or were not part of the training set (a.k.a *out-of-vocabulary* or by its acronym *OOV*). For those cases, the other candidate models cannot give any results because they have a strong requisite that requires the trained model to be used with domain names that were originally part of the training set. If a domain name does not exist or was not part of the training set then the model does not know any vector representation for that domain name, therefore it cannot find any nearest vectors.
But as explained in Section 3.2.2.5, the only requirement that *FastText* needs to meet in order to find a vector representation for an arbitrary domain name is that at least one match exists between any subword of the *OOV* word and any subword used during the training phase (by any word in the vocabulary). *Table 22* shows the results that were obtained by our best model based on *FastText* when retrieving the most similar sites to

samtanderuniversidades.con.uy (a domain that does not exist, and it was not part of the training set).

Most similar domain names to *samtanderuniversidades.con.uy* (banking)

Domain name	***Type***	***Cosine distance***	***Observations***
santanderuniversidades.com.uy	banking	0.995	This is the real site
bancamovilsantander.com.uy	banking	0.953	
santander.com.uy	banking	0.918	
multidiscount.net	banking	0.811	
bcu.gub.uy	banking	0.808	
discbank.com.uy	banking	0.750	
browserforthebetter.com	-	0.785	Domain does not exist anymore (Feb-2019)
brou.com.uy	banking	0.751	
nbc.com.uy	banking	0.749	Domain does not exist anymore (Feb-2019)

Table 22 - most similar sites found by our FastText based model for an oov domain name.

As we can see in *Table 22* our *FastText* based model is able to find similar sites for a domain name that does not really exist (and was not part of the training). Although a formal study of the threshold value would be required, it looks like a carefully selected threshold could be helpful for identifying domain names that for some reason are incorrect, and also to find the correct match for it. A domain name could be bad formed because of many reasons, for example because it was typed incorrectly with a typo or because a harmful software shows a bad formed url intentionally (for example *typosquatted domains*[63] or *IDN homograph attacks*[64]) trying to

[63] https://en.wikipedia.org/wiki/Typosquatting
[64] https://en.wikipedia.org/wiki/IDN_homograph_attack

deceive a user to redirect him/her to a website that looks identically to the original one but generally designed to steal user credentials, banking and credit card details (a.k.a *phishing*). At the moment of writing this, the *Google Chrome* browser is experimenting with a new feature (*Figure 38*) that tries to identify these kind of risky urls and generate suggestions about possible desired sites. This evidences that having reliable methods for identifying fraudulent sites is a hot-topic right now.

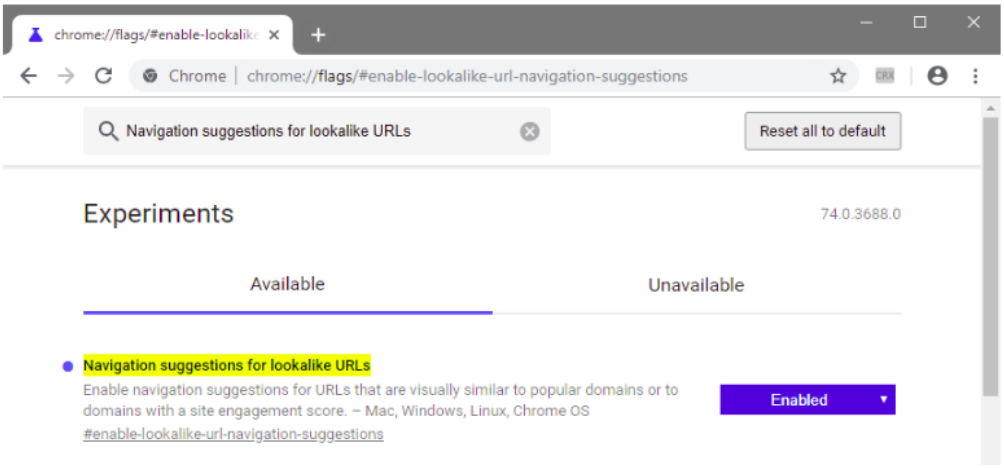


Figure 38 - Experimental feature in Google Chrome that warns about suspicious urls
(image credits: ZDNet[65]).

Hence, one possible application of our DNS vector space model (*DNS-VSM*) is related to security as well as user experience by suggesting potential matches for sites that look like other well known domain names. We think that our model could be combined with some string distance metric (for example the *Levenshtein distance*[66]) to improve the detection of fraudulent sites.

We also see a potential application of our model in a *parental-control* system, filtering (automatically) risky content or adult specific content. For example, one could save a short list of domain names to be blocked (even one or two sites would be enough) and then, the system could find the most similar domain names to those sites and automatically add them to the blacklist (for those results with a high confidence). By doing this the

[65] https://www.zdnet.com/article/google-chrome-to-get-warnings-for-lookalike-urls/
[66] https://en.wikipedia.org/wiki/Levenshtein_distance

system would increase and handle the blacklist automatically. As an example, in *Table 23* we can see the results that are obtained when asking to our *FastText* based model for the most similar sites to *pornhub.com*. After studying a good threshold the system could add some of these results to the blacklist and repeat the process finding more sites to block until no more sites are found upper the threshold.
Other natural application of our *DNS-VSM* is in the recommender systems area, by providing a core block for finding similar sites to the sites that are generally navigated by the user. In the recommender systems area this kind of recommender falls in the category of *content-based* recommenders. The approach behind this kind of recommender systems is to represent system items (products, movies, books, etc) by their main properties (title, price, category, etc) and then, similar items to those that the user liked in the past (for example those items that were bought, visited, or rated positive by the user) are recommended. The reader can see [101] for a good summary about this and other recommender systems techniques.

Domain name	***Type***	***Cosine distance***
youporn.com	adult website	0.879
phncdn.com	adult website	0.84
tube8.com	adult website	0.795
youporn.com.es	adult website	0.758
videospornhub.com	adult website	0.708
xxxcupid.com	adult website	0.696
german-youporn.com	adult website	0.696
pornhubpremium.com	adult website	0.693
genericlink.com	-	0.687
youporngay.com	adult website	0.68

Table 23 - most similar sites to pornhub.com (an adult specific content site).

Besides the potential use cases just mentioned, we can think many other applications that could benefit from the usage of our *DNS-VSM*. Clickstream analysis, representation and clustering of users navigation profiles, competitive analysis, optimization of cache systems in recursive DNS resolvers, and the list grows up.

For this reason, as a contribution to the research community we are releasing a set of vectors of the *DNS-VSM* (trained on a similar dataset to the one used in this thesis), which we made available for download through the github page in [1]. With this, we hope that further work can be done using these vectors, for example evaluating the *DNS-VSM* in the task of analogical reasoning (using a specific evaluation set created for this task), how relationships between domain names in Internet have evolved through time (our vectors gives a picture at 2013 and since then many domains have disappeared and many others have been created), or creating new applications on top of them.

Part Ⅲ
CONCLUSIONS

5. Chapter V - Conclusions and future work

Knowing semantic information about Internet domain names is something crucial for many engineering activities, with practical application in many areas.
Common uses cases that we can see nowadays include websites recommendations based on similar sites or competitive analysis (for example *Alexa*[67]*, SImilarWeb*[68] *or Google Similar Pages*[69]), but many others applications have been identified and proposed in this work such as identification of fraudulent or risky sites, parental-control system, *UX* improvements (based on recommendations, spell correction, etc), click-stream analysis, representation and clustering of users navigation profiles, competitive analysis, optimization of cache systems in recursive DNS resolvers, and more.

Current solutions and strategies to identify similarities between Internet domain names fall mainly in two categories: *client-side component based*, or *content-indexing based*. Both kind of solutions have lot of disadvantages. Solutions that use client-side components (generally a browser plugin) require execution permissions that not all users are willing to give, they are intrusive and comprise user confidentially. They are difficult to deliver (users need to be motivated and convinced in order to install the extension) and they are not representative of a global audience. Furthermore, browser plugins do not work well in all mobile devices. On the other hand, solutions that use a content-indexing strategy need to scan and classify the content in predefined set of topics (generally topics are not learned automatically), requiring the content to use good metadata for topic discovery (note that contents and its metadata are not owned by the similarity system), being difficult, expensive, and time consuming to be implemented at a web scale.

[67] https://www.alexa.com/
[68] https://www.similarweb.com/
[69] https://chrome.google.com/webstore/detail/google-similar-pages/pjnfggphgdjblhfjaphkjhfpiiekbbej

In this work, a novel approach to address the problem of finding similarities between Internet domain names (without suffering from the previous mentioned disadvantages) is presented. The solution analyzes real anonymized *DNS* log queries from a big amount of *DNS* log files which come from recursive *DNS* servers from a large Internet Service Provider (*ISP*) in Uruguay.
The fundamentals ingredients behind the solution take many ideas from linguistics and the *NLP* field. In particular, the proposed solution is strongly motivated by the *distributional hypothesis* from linguistics, which suggests that words that are used and occur in the same contexts tend to purport similar meanings [33].

A simple yet effective trick that gives the foundations of this work consists of mapping the concept of *words* and *contexts* to the *DNS* scenario in this way: a single word in a text document corresponds to a domain name in a *DNS* log file, and the context (neighbors) words corresponds to the domain names that were queried close (for example in a fixed length time window or in a same web user session).
While working on this mapping, a set of characteristics and limitations about the *DNS* data and how the *DNS* system works are identified (noise in *DNS* traces associated to record types that are not web navigation related, *TTL* in *DNS* resolvers, many users behind *NAT* enabled gateways, dynamic IP addresses, background traffic not triggered by users, among others) and addressed before applying any *NLP* algorithm to our problem.

Hence, a main contribution of this work is presented in Section 4.2 where a detailed preprocessing pipeline with specific steps (*DNS* record type filter, service type filter, simplification of subdomain, removal of top queried domains and well-known applications domain names, IP grouping, removal of automatic requests, simplification in the navigation path, split of long traces based on time window) is defined to move the original problem to a problem in the *NLP* field.

As a second contribution, Sections 4.4, 4.5 and 4.6 show that once the preprocessing pipeline is applied and the *DNS* log files are transformed to a standard text corpus in the *NLP* field then, *state-of-the art* techniques for *word embeddings* such as *Word2Vec* (with *Skip-Gram* and *CBOW* architectures), *App2Vec* (adding a weighting factor based on time

gaps between domain names) and *FastText* (adding sub-word level information) can be successfully applied to the corpus in order to build what we called a *DNS-VSM* (a vector space model for domain names). In our *DNS-VSM* domain names are represented by vectors (a.k.a *embeddings*) with 64 dimensions with the main characteristic that semantically related domain names are mapped to nearby points in the high dimensional space. This *DNS-VSM* is built only using information of *DNS* queries without any other previous knowledge about the content of those domains.

Through visual inspection we are able to confirm that all these candidates models show good results in the task of finding semantically related domain names. In order to choose the best model, in Section 4.7 a formal comparison of these candidate models is carried out, showing that the *FastText* based model (with *Skip-Gram* architecture and *n-grams* range between 11 and 17) outperforms considerably the other candidate models as well as the baseline models (*random guessing* and *popularity based*, see Section 4.3.1).
In particular, the evaluation of the *MAP@k* metric (see Section 4.3) shows that the *FastText* based model is 10.5%, 17.8%, 17.8% and 435.5% superior than *Word2Vec* with *Skip-Gram*, *App2Vec, Word2Vec* with *CBOW* and the best baseline model (*popularity based*) respectively.

As part of the comparison, the *recall*, *precision* and *f1-score* metrics presented in Section 4.3 are evaluated. In all the experiments similar results are obtained, being *FastText* the best, followed by *Word2Vec* with *Skip-Gram* which is also better than *App2Vec* and *Word2Vec* with *CBOW.* Regarding these last two models and according to our experiments, there is no strong evidence to ensure that *App2Vec* is better than *Word2Vec* with *CBOW* in our specific scenario. Nevertheless, an advantage that was observed of using *App2Vec* over *Word2Vec* with *CBOW* is related to the training time, being *App2Vec* faster to get good results during the first epochs.

In regards to the best candidate model based on *FastText*, different *n-gram* ranges are evaluated using different evaluation criterias. The optimal configuration is found using *min-ngram = 11* and *max-ngram = 17.* For this configuration, the highest *MAP@k* value obtained is 0.238 (with *k*

= 1 and k_{actual} = 10) and the highest *F1-score* is 0.144 (with k = 10 and k_{actual} = 10).
It is important to note that usually when working with standard text corpus the best results are obtained with much shorter *n-grams* of about 3 to 6 characters. In our case this very short *n-grams* are not very useful because they encompass words like *.www*, *.com or .com.uy* (among others) that do not help to determine the compatibility between different domains. By using longer *n-grams* we are able to include longer substrings of the domain names that are more adequate to determine whether two domains are similars or not.

The last contribution of this work is a set of vectors of the *DNS-VSM* (trained on a similar dataset to the one used in this thesis), which we made available for download through the github page in [1]. With this, we hope that further work can be done using these vectors.

For example, some possible directions of future work with the *DNS-VSM* could be related to the creation of an evaluation set for the task of analogical reasoning, the analysis of how relationships between domain names in Internet have evolved through time (our vectors give a picture at 2013 and since then many domains have disappeared and many others have been created), or the development of new applications on top of the *DNS* embeddings.
Additionally, an interesting extension of this work could evaluate other word embeddings techniques such as *GloVe* (see Section 3.2.3), *Swivel [102]* or the recently published (at the moment of writing this) contextual word embedding techniques *ELMo [93]* and *BERT [94]*.
Although in this work we focus in embeddings to the word level, in Chapter 3 some basic techniques for document embeddings are presented. But other more sophisticated for different document sizes exist, for example *Skip-Thought Vectors [81]*, or *Doc2vec [82]*. Good opportunities of future research exist regarding the usage of these techniques in order to analyze user's traces as sentences or paragraphs (composed of domain names instead of words). Then, those sentences and paragraph could be aggregated in some way to build a document to represent the user's navigation profile. Clustering of users based on their navigation profiles is a high valuable information for any telecom company.

6. Bibliography

[1] W. Lopez, “Vector Space Model for DNS,” DNS-VSM, Mar-2019. [Online]. Available: https://github.com/dns-vsm/embeddings.

[2] Ofcom, “Adults’ media use and attitudes. report 2016,” Apr-2016. [Online]. Available: https://www.ofcom.org.uk/__data/assets/pdf_file/0026/80828/2016-adults-media-use-and-attitudes.pdf.

[3] J. Yan, N. Liu, G. Wang, W. Zhang, Y. Jiang, and Z. Chen, “How much can behavioral targeting help online advertising?,” in Proceedings of the 18th international conference on World wide web - WWW ’09, 2009, pp. 261–270.

[4] A. Ahmed, Y. Low, M. Aly, V. Josifovski, and A. J. Smola, “Scalable distributed inference of dynamic user interests for behavioral targeting,” in Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’11, 2011, pp. 114–122.

[5] H. B. McMahan et al., “Ad click prediction,” in Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’13, 2013, pp. 1222–1230.

[6] P. Norvig, “Statistical learning as the ultimate agile development tool,” in ACM 17th Conference on Information and Knowledge Management Industry Event (CIKM-2008), 2008.

[7] H. Cui, J. Yang, Y. Liu, Z. Zheng, and K. Wu, “Data Mining-based DNS Log Analysis,” Annals of Data Science, vol. 1, no. 3–4, pp. 311–323, 2014.

[8] W. Ruan, Y. Liu, and R. Zhao, “Pattern Discovery in DNS Query Traffic,” Procedia Comput. Sci., vol. 17, pp. 80–87, 2013.

[9] Q. Lai, C. Zhou, H. Ma, Z. Wu, and S. Chen, “Visualizing and characterizing DNS lookup behaviors via log-mining,” Neurocomputing, vol. 169, pp. 100–109, 2015.

[10] M. E. Snyder, R. Sundaram, and M. Thakur, “Preprocessing DNS Log Data for Effective Data Mining,” in 2009 IEEE International Conference on Communications, 2009, pp. 1366–1370.

[11] Network Working Group, “Internet Domain Name System Standard: Domain names - concepts and facilities (RFC 1034),” RFC 1034 - Domain names - concepts and facilities, Nov-1987. [Online]. Available: https://www.ietf.org/rfc/rfc1034.txt.

[12] Network Working Group, “Internet Domain Name System Standard: Domain names - implementation and specification (RFC 1035),” DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION, 1987. [Online]. Available: https://www.ietf.org/rfc/rfc1035.txt.

[13] P. Albitz and C. Liu, DNS and BIND. “O’Reilly Media, Inc.,” 2001.

[14] “SimilarWeb Review,” Seperia. [Online]. Available: https://www.seperia.com/blog/competitive-intelligence-tool-reviews/similarweb/.

[15] “SimilarWeb vs Alexa – Competing on Competitive Intelligence,” Seperia. [Online]. Available:

https://www.seperia.com/blog/competitive-intelligence-tool-reviews/similarweb-vs-alexa/.
[16] “Mobile and tablet internet usage exceeds desktop for first time worldwide,” Statcounter. [Online]. Available: http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide.
[17] D. Jurafsky and J. H. Martin, Speech and Language Processing. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Third Edition draft. .
[18] M. Baroni, G. Dinu, and G. Kruszewski, “Don’t count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors,” Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2014.
[19] J. Pennington, R. Socher, and C. Manning, “Glove: Global Vectors for Word Representation,” in Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014.
[20] S. Li, J. Zhu, and C. Miao, “A Generative Word Embedding Model and its Low Rank Positive Semidefinite Solution,” Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. 2015.
[21] P. D. Turney and P. Pantel, “From Frequency to Meaning: Vector Space Models of Semantics,” 1, vol. 37, pp. 141–188, Feb. 2010.
[22] F. Almeida and G. Xexéo, “Word Embeddings: A Survey,” CoRR, vol. abs/1901.09069, 2019.
[23] D. Bollegala, K. Hayashi, and K.-I. Kawarabayashi, “Learning linear transformations between counting-based and prediction-based word embeddings,” PLoS One, vol. 12, no. 9, p. e0184544, Sep. 2017.
[24] J. E. Alvarez, “A review of word embedding and document similarity algorithms applied to academic text,” BSc, University of Freiburg, 2017.
[25] R. Rehurek, “Scalability of semantic analysis in natural language processing,” 2011.
[26] T. Mikolov, S. Kombrink, L. Burget, J. Cernocky, and S. Khudanpur, “Extensions of recurrent neural network language model,” in 2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2011.
[27] Y. Bengio, H. Schwenk, J.-S. Senécal, F. Morin, and J.-L. Gauvain, “Neural Probabilistic Language Models,” in Studies in Fuzziness and Soft Computing, pp. 137–186.
[28] H. Schwenk, “Continuous space language models,” Comput. Speech Lang., vol. 21, no. 3, pp. 492–518, 2007.
[29] T. Mikolov, “Statistical language models based on neural networks,” Brno University of Technology, 2012.
[30] G. Salton, The SMART retrieval system: experiments in automatic document processing. Prentice Hall, 1971.
[31] G. Salton, A. Wong, and C. S. Yang, “A vector space model for automatic indexing,” Commun. ACM, vol. 18, no. 11, pp. 613–620, 1975.
[32] L. Wittgenstein, Philosophical Investigations. Blackwell, 1953.
[33] Z. S. Harris, “Distributional Structure,” Word World, vol. 10, no. 2–3, pp.

146–162, 1954.
[34] W. N. Locke and A. D. Booth, Machine translation of languages: fourteen essays. 1955.
[35] J. R. Firth, A Synopsis of Linguistic Theory, 1930-1955. 1957.
[36] “Vector Representations of Words | TensorFlow,” TensorFlow. [Online]. Available: https://www.tensorflow.org/tutorials/word2vec. [Accessed: 29-May-2018].
[37] C. D. Manning, P. Raghavan, and H. Schutze, Introduction to Information Retrieval. 2008.
[38] K. W. Church and P. Hanks, “Word association norms, mutual information, and lexicography,” in Proceedings of the 27th annual meeting on Association for Computational Linguistics -, 1989.
[39] J. A. Bullinaria and J. P. Levy, “Extracting semantic representations from word co-occurrence statistics: a computational study,” Behav. Res. Methods, vol. 39, no. 3, pp. 510–526, Aug. 2007.
[40] P. Pantel and D. Lin, “Discovering word senses from text,” in Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’02, 2002.
[41] P. D. Turney and M. L. Littman, “Measuring praise and criticism: Inference of semantic orientation from association,” ACM Transactions on Information Systems, vol. 21, no. 4, pp. 315–346, 2003.
[42] K. Lund and C. Burgess, “Producing high-dimensional semantic spaces from lexical co-occurrence,” Behav. Res. Methods Instrum. Comput., vol. 28, no. 2, pp. 203–208, 1996.
[43] Douglas L. T. Rohde and Laura M. Gonnerman and David C. Plaut, “An improved model of semantic similarity based on lexical co-occurence,” COMMUNICATIONS OF THE ACM, vol. 8, pp. 627–633, 2006.
[44] R. Lebret and R. Collobert, “Word Embeddings through Hellinger PCA,” in Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics, 2014.
[45] D. I. Martin and M. W. Berry, “Mathematical Foundations Behind Latent Semantic Analysis,” in Handbook of Latent Semantic Analysis, .
[46] M. W. Berry and M. Browne, Understanding Search Engines: Mathematical Modeling and Text Retrieval. SIAM, 2005.
[47] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, “Indexing by latent semantic analysis,” Journal of the American Society for Information Science, vol. 41, no. 6, pp. 391–407, 1990.
[48] T. K. Landauer and S. T. Dumais, “A solution to Plato’s problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge,” Psychol. Rev., vol. 104, no. 2, pp. 211–240, 1997.
[49] R. Rapp, “Word Sense Discovery Based on Sense Descriptor Dissimilarity,” Proceedings of the Ninth Machine Translation Summit, pp. 315–322, 2003.
[50] Y. Bengio, R´. Ducharme, P. Vincent, and C. Janvin, “A neural probabilistic language model,” The Journal of Machine Learning Research, vol. 3, pp. 1137–1155, 2003.
[51] A. A. Markov, Example of a Statistical Investigation of the Text of “Eugene Onegin” Illustrating the Dependence Between Samples in Chain. 1955.

[52] J. T. Goodman, “A bit of progress in language modeling, extended version. Technical report MSR-TR-2001-72,” 2001.
[53] K. Wang, C. Thrasher, E. V. Viegas, X. Li, and B.-J. (paul) Hsu, “An overview of Microsoft web N-gram corpus and applications,” in Proceedings of the NAACL HLT 2010 Demonstration Session, 2010, pp. 45–48.
[54] T. Brants and A. Franz, “All our n-gram are belong to you,” Google AI blog, 2006. [Online]. Available: https://ai.googleblog.com/2006/08/all-our-n-gram-are-belong-to-you.html.
[55] D. Graff and C. Cieri, English Gigaword. Linguistic Data Consortium, 2003.
[56] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean, “Efficient estimation of word representations in vector space,” vol. abs/1301.3781, 2013.
[57] T. Mikolov, M. Karafiát, L. Burget, J. Cernocky, and S. Khudanpur, “Recurrent Neural Network Based Language Model,” in INTERSPEECH, 2010.
[58] J. L. Elman, “Finding Structure in Time,” Cogn. Sci., vol. 14, no. 2, pp. 179–211, 1990.
[59] M. Bodén, “A Guide to Recurrent Neural Networks and Backpropagation,” IN THE DALLAS PROJECT, SICS TECHNICAL REPORT T2002:03, SICS, 2002.
[60] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” IEEE Trans. Neural Netw., vol. 5, no. 2, pp. 157–166, 1994.
[61] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean, “Distributed representations of words and phrases and their compositionality,” CoRR, vol. abs/1310.4546, 2013.
[62] T. Mikolov, “Learning Representations of Text using Neural Networks,” presented at the NIPS Deep Learning Workshop 2013, Harrah’s Sand Harbor II room, Lake Tahoe, USA, 09-Dec-2013.
[63] C. McCormick, “Word2Vec Tutorial - The Skip-Gram Model,” Apr-2016. [Online]. Available: http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/.
[64] X. Rong, “word2vec Parameter Learning Explained,” CoRR, vol. abs/1411.2738, 2014.
[65] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, Armand Joulin, “Advances in Pre-Training Distributed Word Representations,” 2017.
[66] Q. Ma, S. Muthukrishnan, and W. Simpson, “App2Vec: Vector modeling of mobile apps and applications,” in 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), 2016.
[67] M. Frederic and B. Yoshua, “Hierarchical probabilistic neural network language model,” in AISTATS, 2005, pp. 246–252.
[68] J. Weston, F. Ratle, and R. Collobert, “Deep learning via semi-supervised embedding,” in Proceedings of the 25th international conference on Machine learning - ICML ’08, 2008.
[69] Y. W. T. Andriy Mnih, “A Fast and Simple Algorithm for Training Neural Probabilistic Language Models,” 29th International Conference on Machine Learning, pp. 1751–1758, 2012.

[70] “fastText.” [Online]. Available: https://fasttext.cc/index.html. [Accessed: 05-Jun-2018].
[71] Piotr Bojanowski and Edouard Grave and Armand Joulin and Tomas Mikolov, “Enriching Word Vectors with Subword Information,” CoRR, vol. abs/1607.04606, 2016.
[72] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, “Bag of Tricks for Efficient Text Classification,” in Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers, 2017.
[73] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jegou, T. Mikolov, “FastText.zip: Compressing text classification models,” CoRR, vol. abs/1612.03651, 2016.
[74] RaRe-Technologies, “RaRe-Technologies/gensim,” GitHub. [Online]. Available: https://github.com/RaRe-Technologies/gensim. [Accessed: 06-Jun-2018].
[75] “Word representations · fastText.” [Online]. Available: https://fasttext.cc/index.html. [Accessed: 06-Jun-2018].
[76] J. Pennington, “GloVe: Global Vectors for Word Representation.” [Online]. Available: https://nlp.stanford.edu/projects/glove/. [Accessed: 03-Jun-2018].
[77] J. Pennington, R. Socher, and C. Manning, “Glove: Global Vectors for Word Representation (Conference Video on Empirical Methods in Natural Language Processing (EMNLP2014)),” 23-Nov-2014. [Online]. Available: https://www.youtube.com/watch?v=RyTpzZQrHCs. [Accessed: 03-Jun-2018].
[78] “Learning the meaning behind words,” Google Open Source Blog. [Online]. Available: https://opensource.googleblog.com/2013/08/learning-meaning-behind-words.html. [Accessed: 03-Jun-2018].
[79] Mikolov, T and Yih, W.-T and Zweig, G, “Linguistic Regularities in Continuous Space Word Representations,” in Proceedings of NAACL-HLT, 2013, pp. 746–751.
[80] Aleksandr Drozd and Anna Gladkova and Satoshi Matsuoka, “Word Embeddings, Analogies, and Machine Learning: Beyond King - M an + W oman = Queen,” in COLING, 2016.
[81] R. Kiros et al., “Skip-Thought Vectors,” NIPS, 2015.
[82] Q. V. Le and T. Mikolov, “Distributed Representations of Sentences and Documents,” ICML’14 Proceedings of the 31st International Conference on International Conference on Machine Learning, vol. 32, pp. II–1188–II–1196, 2014.
[83] H. P. Luhn, “A Statistical Approach to Mechanized Encoding and Searching of Literary Information,” IBM Journal of Research and Development, vol. 1, no. 4. pp. 309–317, 1957.
[84] K. S. Jones, “A STATISTICAL INTERPRETATION OF TERM SPECIFICITY AND ITS APPLICATION IN RETRIEVAL,” Journal of Documentation, vol. 28, no. 1. pp. 11–21, 1972.
[85] H. P. Luhn, “The Automatic Creation of Literature Abstracts,” IBM Journal of Research and Development, vol. 2, no. 2. pp. 159–165, 1958.
[86] S. E. Robertson and K. Sparck Jones, “Relevance weighting of search

terms," Journal of the American Society for Information Science, vol. 27, no. 3. pp. 129–146, 1976.
[87] C. T. Yu and G. Salton, "Precision Weighting---An Effective Automatic Indexing Method," Journal of the ACM, vol. 23, no. 1. pp. 76–88, 1976.
[88] G. Amati and C. J. Van Rijsbergen, "Probabilistic models of information retrieval based on measuring the divergence from randomness," ACM Transactions on Information Systems, vol. 20, no. 4. pp. 357–389, 2002.
[89] S. C. Deerwester et al., "Computer information retrieval using latent semantic structure," 1988.
[90] H. Schutze, "Dimensions of meaning," Proceedings Supercomputing '92. .
[91] W. Pottenger and A. Kontostathis, "Detecting Patterns in the LSI Term-Term Matrix," 2002.
[92] C. E. Shannon, "A Mathematical Theory of Communication," Bell System Technical Journal, vol. 27, pp. 379–423, 1948.
[93] M. Peters et al., "Deep Contextualized Word Representations," in Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), 2018.
[94] J. Devlin, M.-W. Chang, and L. K. T. Kristina, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," CoRR, 2018.
[95] W. Lopez, J. Merlino, and P. Rodriguez-Bocca, "Vector representation of internet domain names using a word embedding technique," 2017 XLIII Latin American Computer Conference (CLEI). 2017.
[96] S. Robertson, "A new interpretation of average precision," in Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '08, 2008.
[97] G. Tsoumakas and I. Katakis, "Multi-Label Classification," in Database Technologies, pp. 309–319.
[98] L. J. P. V. D. Maaten and G. E. Hinton, "Visualizing data using t-SNE," Journal of Machine Learning Research, vol. 9, pp. 2579–2605, 2008.
[99] R. Rehurek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in LREC 2010 Workshop on New Challenges for NLP Frameworks, Malta, 2010.
[100] W. Lopez, "How complex is a given software development task?," Predictions with story points, 2018. [Online]. Available: https://waljoel.github.io/story-points-prediction/Story_Points_Prediction_Demo_v0.9.html.
[101] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, Recommender Systems Handbook. Springer Science & Business Media, 2010.
[102] N. Shazeer, R. Doherty, C. Evans, and C. Waterson, "Swivel: Improving Embeddings by Noticing What's Missing," CoRR, 2016.

7. Appendix A: Final metrics

Table 24 shows the final *MAP@k* metric for the different candidate models and combinations of *k* (number of ordered similar sites considered by the model) and k_{actual} (number of similar sites retrieved from the *Alexa*'s service)

MAP@k \ Model	***Word2Vec (cbow)***	***Word2Vec (skip-gram)***	***App2Vec***	***FastText***
k=1, kactual=1	0.074	0.076	0.073	0.087
k=1, kactual=2	0.106	0.111	0.106	0.126
k=1, kactual=3	0.129	0.137	0.129	0.152
k=1, kactual=5	0.162	0.174	0.159	0.188
k=1, kactual=10	0.21	0.223	0.203	0.238
k=2, kactual=1	0.096	0.099	0.097	0.111
k=2, kactual=2	0.073	0.076	0.074	0.087
k=2, kactual=3	0.093	0.099	0.094	0.11
k=2, kactual=5	0.121	0.13	0.12	0.144
k=2, kactual=10	0.162	0.176	0.158	0.19
k=3, kactual=1	0.107	0.109	0.108	0.123
k=3, kactual=2	0.084	0.088	0.085	0.099
k=3, kactual=3	0.073	0.077	0.074	0.087
k=3, kactual=5	0.097	0.105	0.097	0.117
k=3, kactual=10	0.134	0.147	0.132	0.159
k=5, kactual=1	0.117	0.119	0.118	0.133
k=5, kactual=2	0.096	0.1	0.097	0.113
k=5, kactual=3	0.086	0.09	0.086	0.102
k=5, kactual=5	0.071	0.077	0.071	0.086
k=5, kactual=10	0.103	0.112	0.102	0.123
k=10, kactual=1	0.126	0.129	0.127	0.144
k=10, kactual=2	0.108	0.113	0.109	0.127

k=10, kactual=3	0.099	0.106	0.1	0.118
k=10, kactual=5	0.087	0.094	0.087	0.105
k=10, kactual=10	0.067	0.075	0.067	0.082

Table 24 - Final MAP@k metric

Final average *MAP@k* for Word2Vec (with cbow): **0.107**
Final average *MAP@k* for Word2Vec (with skip-gram): **0.114**
Final average *MAP@k* for App2Vec: **0.107**
Final average *MAP@k* for fastText: 0.126

Table 25 shows the final *F1-score* metric (harmonic mean between *precision* and *recall*, being the *F1-score* better when *precision* and *recall* are higher) for the different candidate models and combinations of *k* (number of ordered similar sites considered by the model) and k_{actual} (number of similar sites retrieved from the *Alexa*'s service)

F1 \ Model	***Word2Vec (cbow)***	***Word2Vec (skip-gram)***	***App2Vec***	***FastText***
k=1, kactual=1	0.074	0.076	0.073	0.087
k=1, kactual=2	0.071	0.074	0.071	0.084
k=1, kactual=3	0.065	0.069	0.065	0.076
k=1, kactual=5	0.054	0.058	0.053	0.063
k=1, kactual=10	0.038	0.041	0.037	0.043
k=2, kactual=1	0.079	0.081	0.08	0.09
k=2, kactual=2	0.088	0.092	0.089	0.104
k=2, kactual=3	0.088	0.094	0.089	0.104
k=2, kactual=5	0.0813	0.087	0.08	0.096
k=2, kactual=10	0.063	0.068	0.061	0.073
k=3, kactual=1	0.075	0.076	0.077	0.085
k=3, kactual=2	0.092	0.096	0.094	0.107
k=3, kactual=3	0.098	0.104	0.099	0.114
k=3, kactual=5	0.096	0.104	0.097	0.113

k=3, kactual=10	0.0798	0.087	0.079	0.093
k=5, kactual=1	0.065	0.066	0.066	0.072
k=5, kactual=2	0.09	0.092	0.09	0.103
k=5, kactual=3	0.103	0.107	0.103	0.119
k=5, kactual=5	0.11	0.117	0.11	0.129
k=5, kactual=10	0.102	0.11	0.102	0.118
k=10, kactual=1	0.048	0.049	0.048	0.054
k=10, kactual=2	0.075	0.079	0.075	0.085
k=10, kactual=3	0.093	0.099	0.094	0.107
k=10, kactual=5	0.112	0.121	0.113	0.131
k=10, kactual=10	0.123	0.135	0.125	0.144

Table 25 - *Final F1-score*

Final average *F1-score* for Word2Vec (with cbow): **0.083**
Final average *F1-score* for Word2Vec (with skip-gram): **0.087**
Final average *F1-score* for App2Vec: **0.083**
Final average *F1-score* for FastText: 0.096

Table 26 shows the final *recall* metric for the different candidate models and combinations of *k* (number of ordered similar sites considered by the model) and k_{actual} (number of similar sites retrieved from the *Alexa*'s service)

Recall \ Model	***Word2Vec (cbow)***	***Word2Vec (skip-gram)***	***App2Vec***	***FastText***
k=1, kactual=1	0.074	0.076	0.073	0.087
k=1, kactual=2	0.053	0.055	0.053	0.063
k=1, kactual=3	0.043	0.046	0.043	0.051
k=1, kactual=5	0.032	0.035	0.032	0.038
k=1, kactual=10	0.021	0.022	0.02	0.024

k=2, kactual=1	0.118	0.121	0.121	0.135
k=2, kactual=2	0.088	0.092	0.089	0.104
k=2, kactual=3	0.073	0.078	0.074	0.087
k=2, kactual=5	0.057	0.061	0.056	0.067
k=2, kactual=10	0.038	0.041	0.037	0.044
k=3, kactual=1	0.151	0.153	0.154	0.171
k=3, kactual=2	0.115	0.12	0.117	0.134
k=3, kactual=3	0.098	0.104	0.099	0.114
k=3, kactual=5	0.076	0.083	0.078	0.091
k=3, kactual=10	0.052	0.057	0.051	0.06
k=5, kactual=1	0.196	0.197	0.197	0.217
k=5, kactual=2	0.157	0.161	0.158	0.18
k=5, kactual=3	0.137	0.142	0.137	0.159
k=5, kactual=5	0.11	0.117	0.11	0.129
k=5, kactual=10	0.076	0.083	0.076	0.088
k=10, kactual=1	0.262	0.271	0.266	0.298
k=10, kactual=2	0.224	0.236	0.226	0.256
k=10, kactual=3	0.202	0.214	0.203	0.233
k=10, kactual=5	0.168	0.181	0.169	0.197
k=10, kactual=10	0.123	0.135	0.125	0.144

Table 26 - Final recall metric

Final average *Recall* for Word2Vec (with cbow): **0.11**
Final average *Recall* for Word2Vec (with skip-gram): **0.115**
Final average *Recall* for App2Vec: **0.111**
Final average *Recall* for FastText: 0.127

Table 27 shows the final *precision* metric for the different candidate models and combinations of *k* (number of ordered similar sites considered by the model) and k_{actual} (number of similar sites retrieved from the *Alexa*'s service)

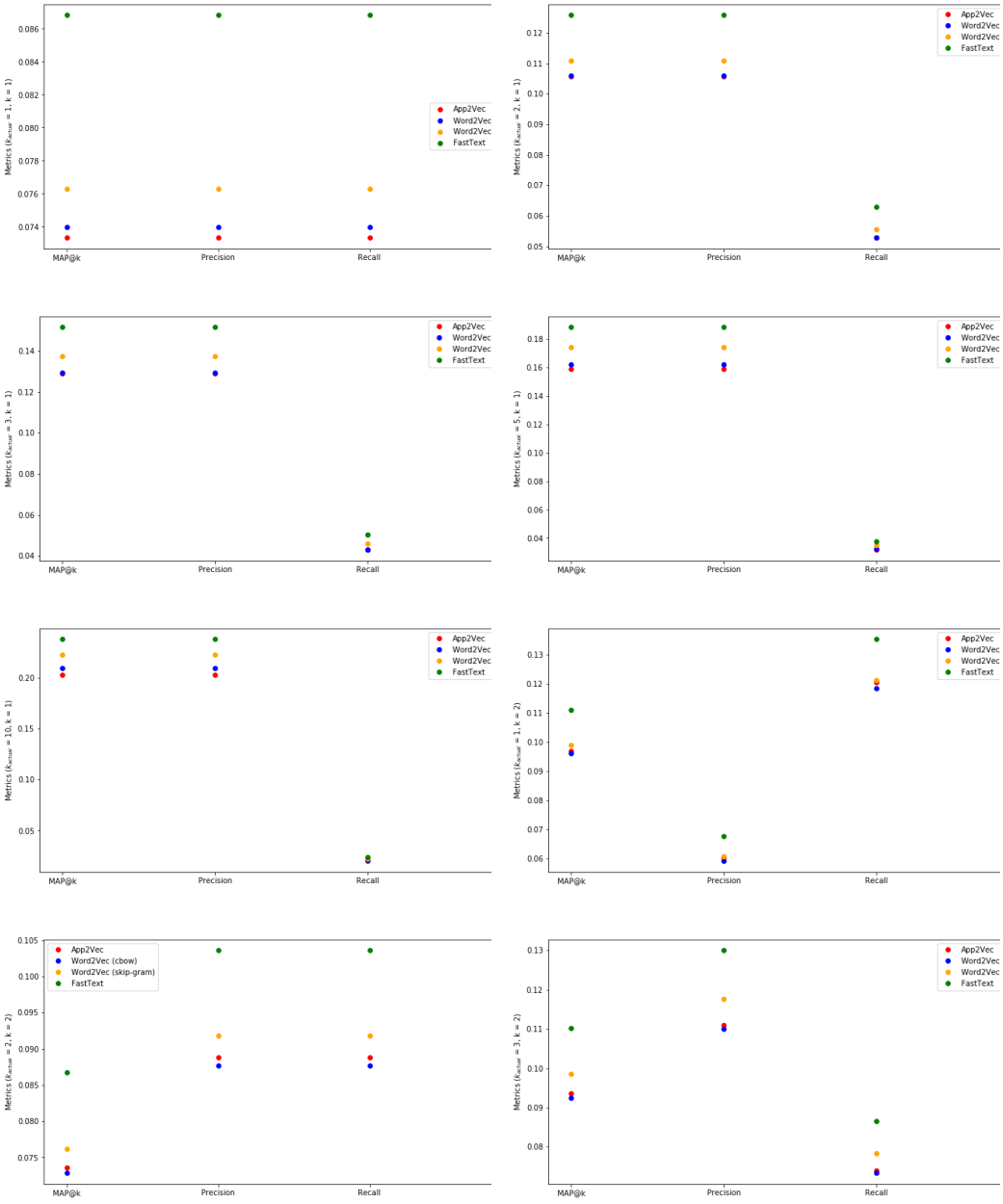
Precision \ Model	***Word2Vec (cbow)***	***Word2Vec (skip-gram)***	***App2Vec***	***FastText***

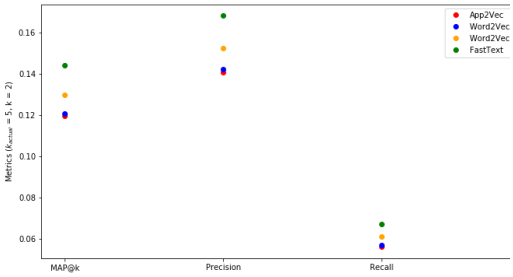
k=1, kactual=1	0.074	0.076	0.073	0.087
k=1, kactual=2	0.106	0.111	0.106	0.126
k=1, kactual=3	0.129	0.137	0.129	0.152
k=1, kactual=5	0.162	0.174	0.159	0.188
k=1, kactual=10	0.21	0.223	0.203	0.238
k=2, kactual=1	0.059	0.061	0.06	0.068
k=2, kactual=2	0.088	0.092	0.089	0.104
k=2, kactual=3	0.11	0.118	0.111	0.13
k=2, kactual=5	0.142	0.153	0.141	0.168
k=2, kactual=10	0.189	0.205	0.184	0.218
k=3, kactual=1	0.05	0.051	0.051	0.057
k=3, kactual=2	0.077	0.08	0.078	0.089
k=3, kactual=3	0.073	0.077	0.074	0.087
k=3, kactual=5	0.127	0.139	0.129	0.151
k=3, kactual=10	0.173	0.051	0.171	0.201
k=5, kactual=1	0.039	0.039	0.039	0.043
k=5, kactual=2	0.063	0.065	0.063	0.072
k=5, kactual=3	0.082	0.085	0.082	0.095
k=5, kactual=5	0.11	0.117	0.11	0.129
k=5, kactual=10	0.153	0.165	0.153	0.177
k=10, kactual=1	0.026	0.027	0.027	0.03
k=10, kactual=2	0.045	0.047	0.045	0.051
k=10, kactual=3	0.06	0.064	0.061	0.07
k=10, kactual=5	0.084	0.09	0.085	0.099
k=10, kactual=10	0.123	0.135	0.125	0.144

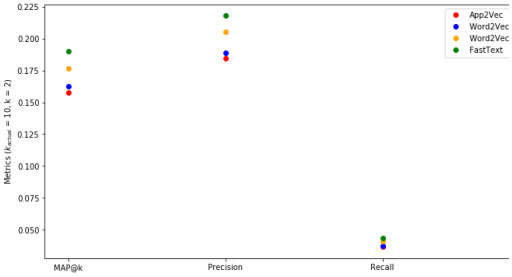
Table 27 - Final precision metric

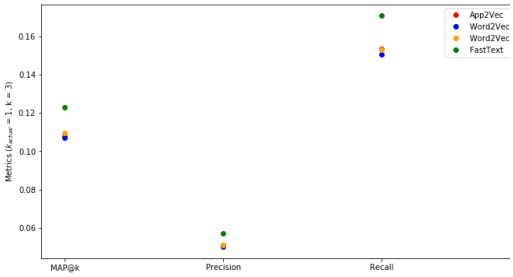
Final average *Precision* for Word2Vec (with cbow): **0.102**
Final average *Precision* for Word2Vec (with skip-gram): **0.103**
Final average *Precision* for App2Vec: **0.102**
Final average *Precision* for FastText: 0.119

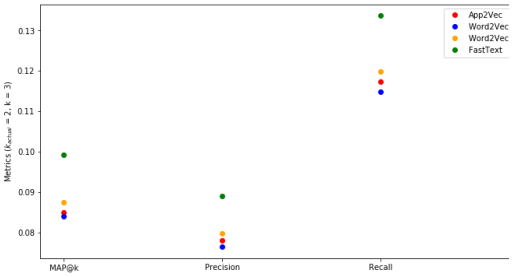
The following charts summarize the values from the previous tables and are helpful for a relative comparison through visual inspection for the quality of the different candidate models.

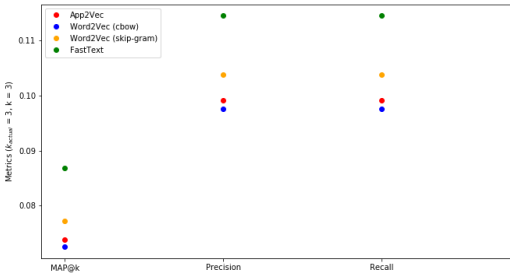


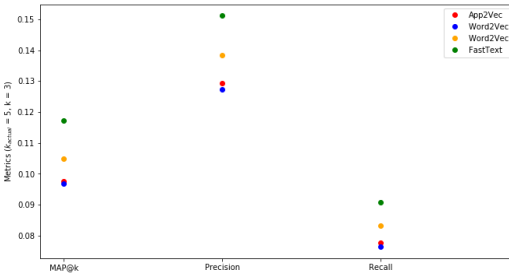

App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 5, k = 2)
MAP@k
Precision
Recall

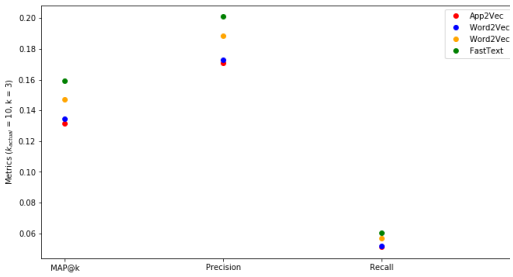

App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 10, k = 2)
MAP@k
Precision
Recall

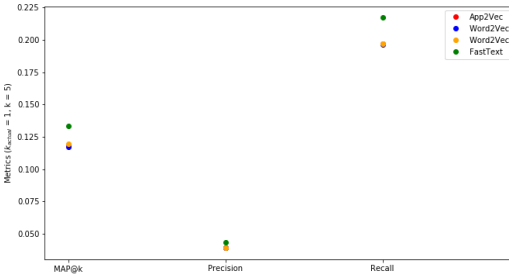

App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 1, k = 3)
MAP@k
Precision
Recall

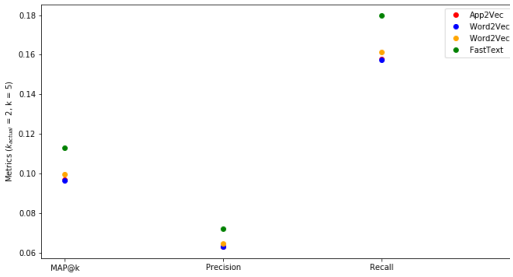

App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 2, k = 3)
MAP@k
Precision
Recall

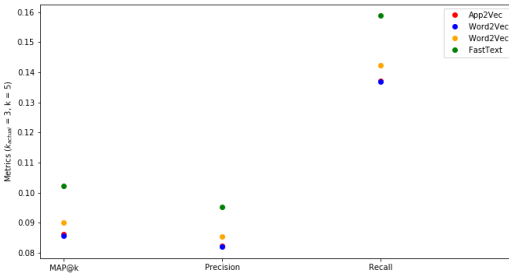

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
Metrics (k_{actual} = 3, k = 3)
MAP@k
Precision
Recall

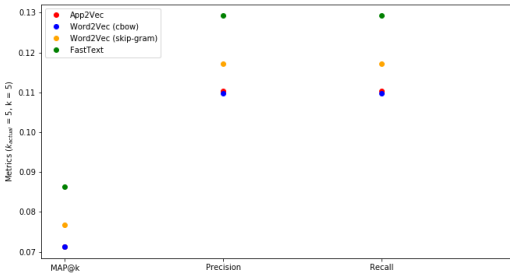

App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 5, k = 3)
MAP@k
Precision
Recall

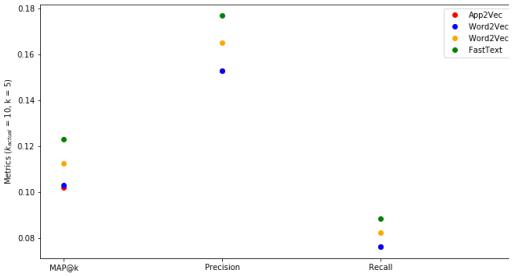

App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 10, k = 3)
MAP@k
Precision
Recall

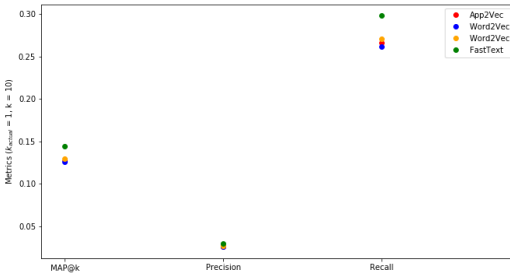

App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 1, k = 5)
MAP@k
Precision
Recall

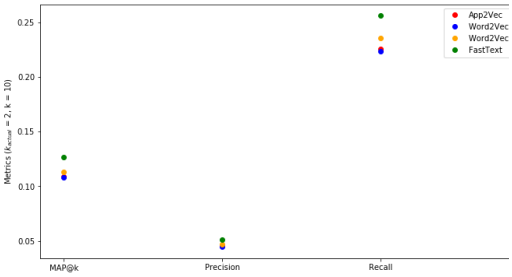

App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 2, k = 5)
MAP@k
Precision
Recall

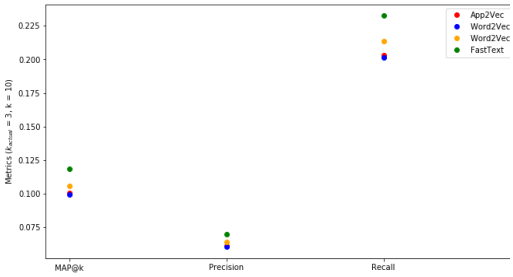

App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 3, k = 5)
MAP@k
Precision
Recall

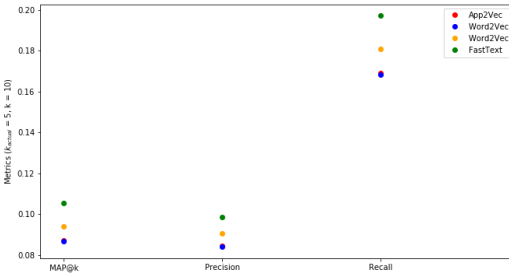

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
Metrics (k_{actual} = 5, k = 5)
MAP@k
Precision
Recall

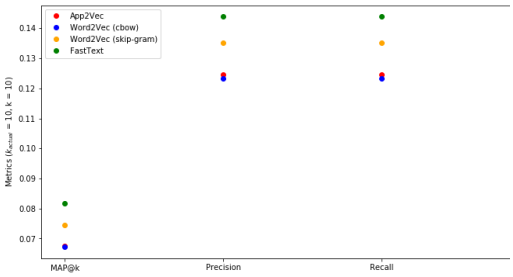

App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 10, k = 5)
MAP@k
Precision
Recall


App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 1, k = 10)
MAP@k
Precision
Recall


App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 2, k = 10)
MAP@k
Precision
Recall

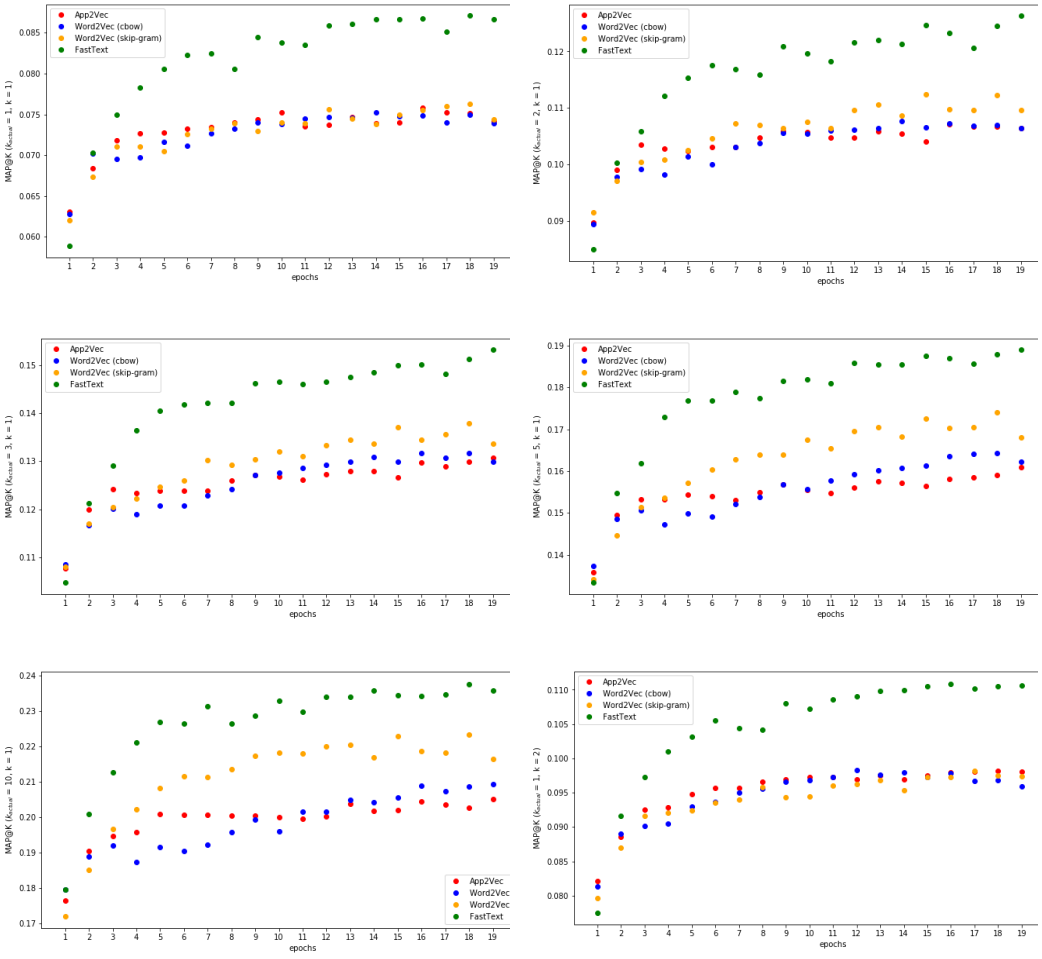

App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 3, k = 10)
MAP@k
Precision
Recall

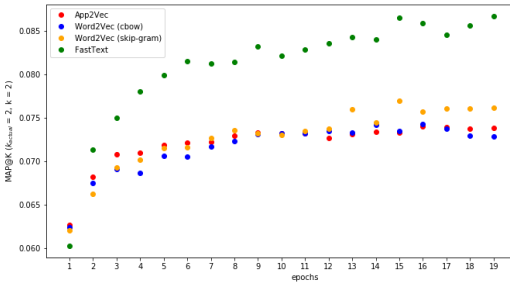

App2Vec
Word2Vec
Word2Vec
FastText
Metrics (k_{actual} = 5, k = 10)
MAP@k
Precision
Recall

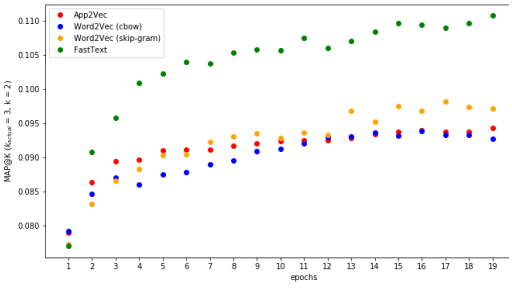

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
Metrics (k_{actual} = 10, k = 10)
MAP@k
Precision
Recall

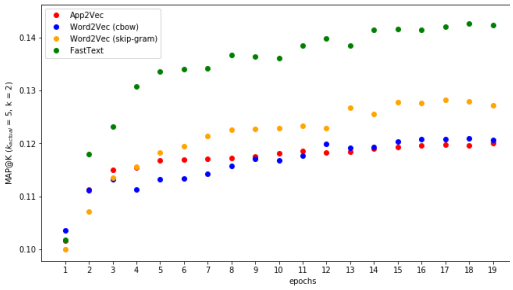
8. Appendix B: Evolution of metrics during training

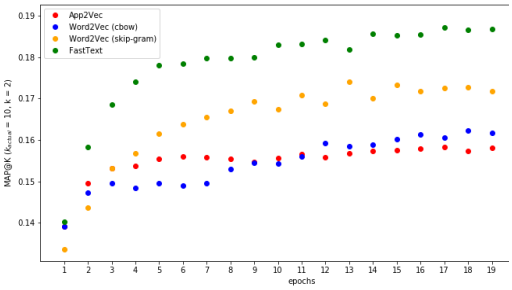
The following charts show the evolution of the *MAP@k* metric during the training phase for the different candidate models and combinations of *k* (number of ordered similar sites considered by the model) and k_{actual} (number of similar sites retrieved from the *Alexa*'s service)

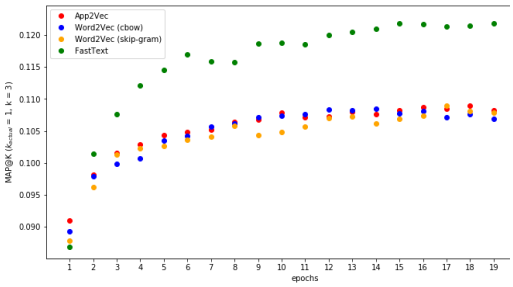


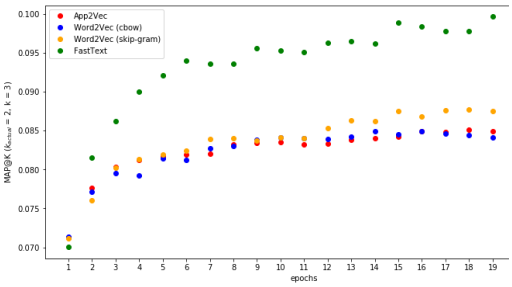

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 2, k = 2)
epochs

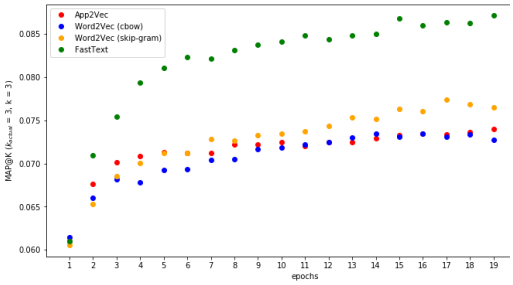

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 3, k = 2)
epochs

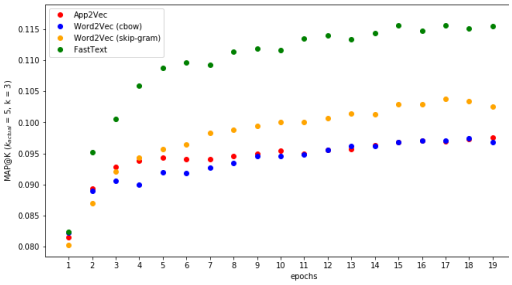

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 5, k = 2)
epochs

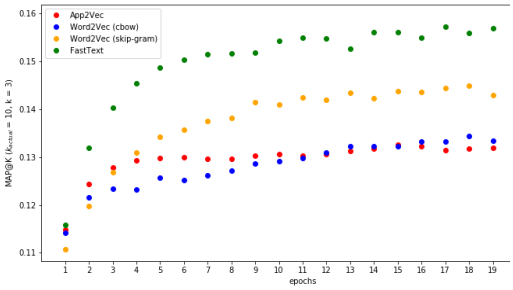

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 10, k = 2)
epochs

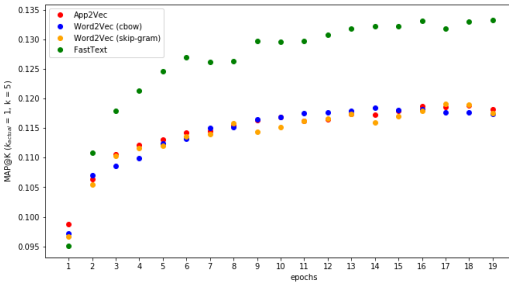

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 1, k = 3)
epochs

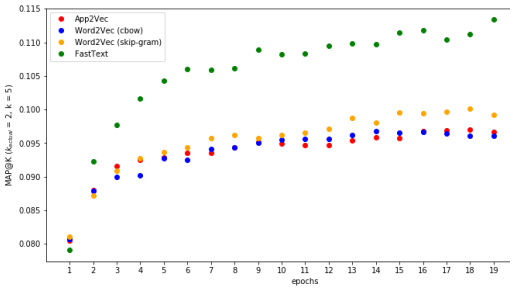

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 2, k = 3)
epochs

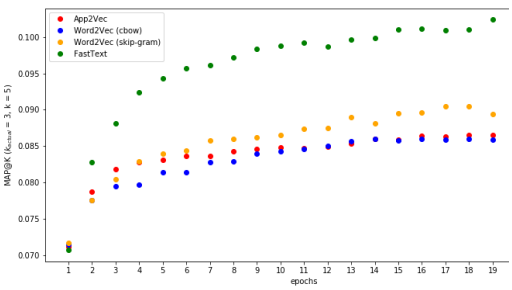

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 3, k = 3)
epochs

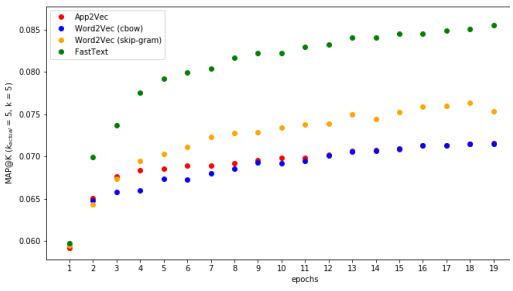

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 5, k = 3)
epochs

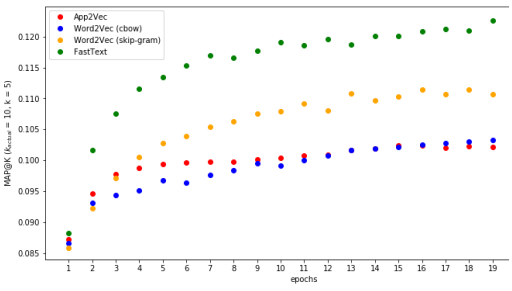

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 10, k = 3)
epochs

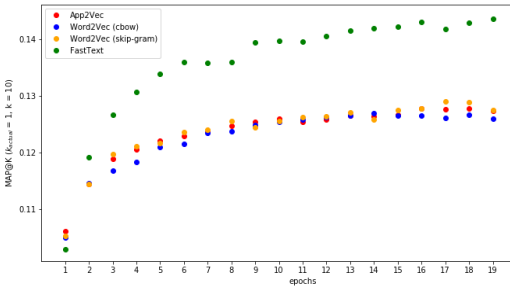

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 1, k = 5)
epochs

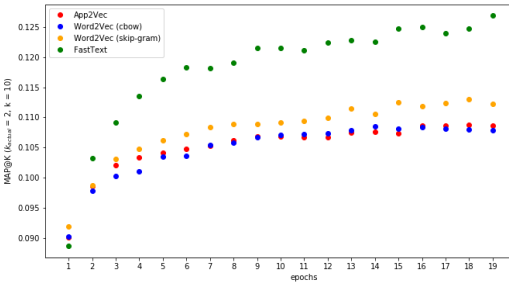

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 2, k = 5)
epochs

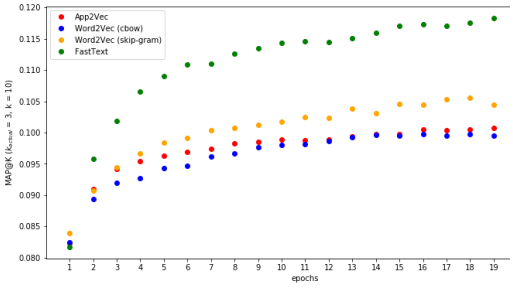

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 3, k = 5)
epochs

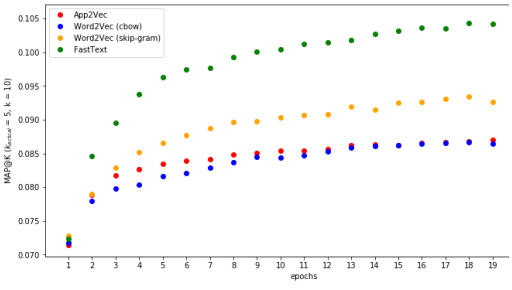

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 5, k = 5)
epochs

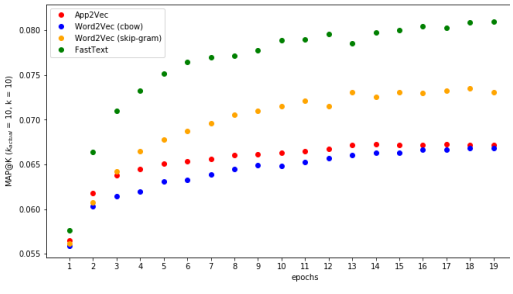

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 10, k = 5)
epochs


App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 1, k = 10)
epochs

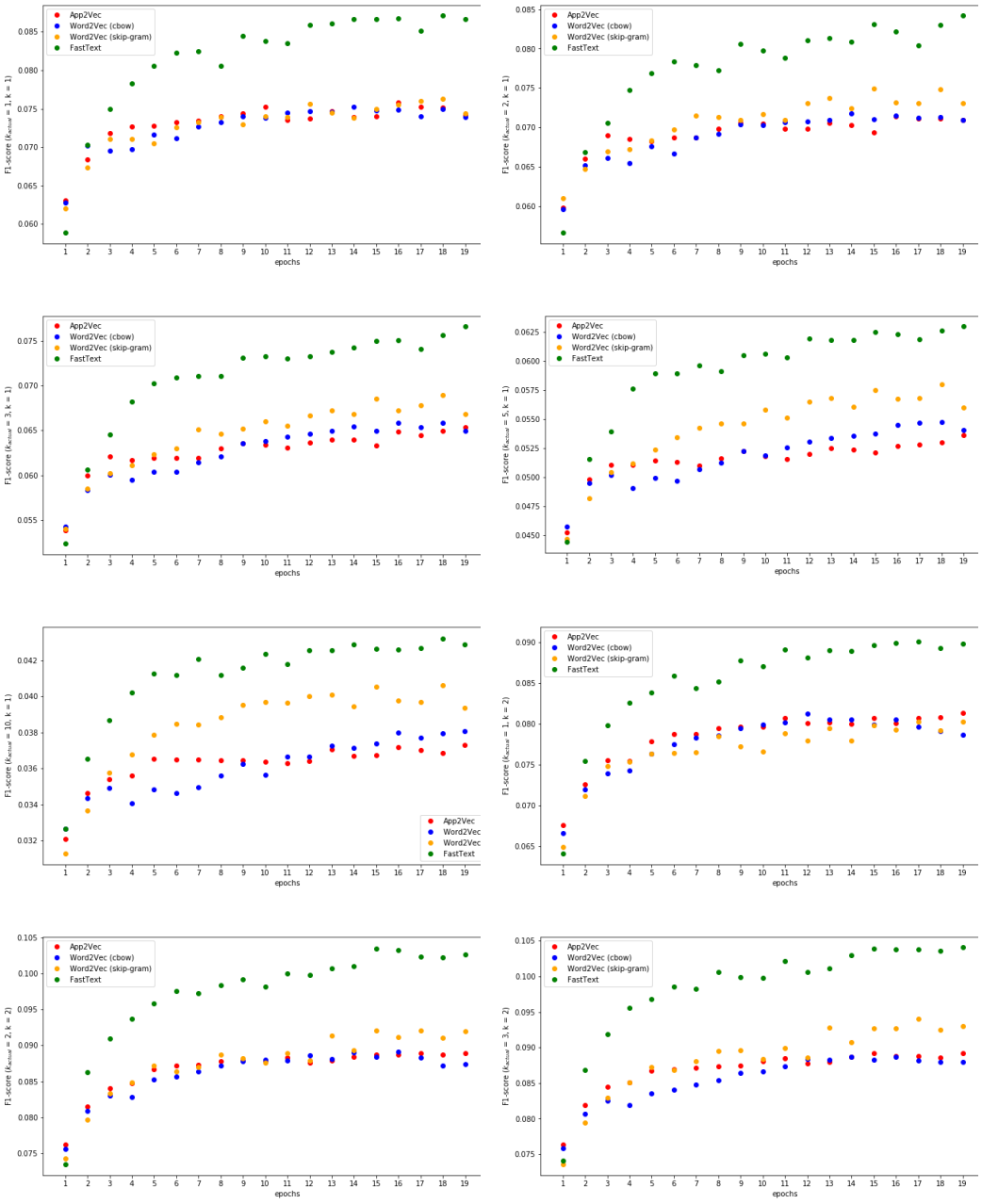

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 2, k = 10)
epochs

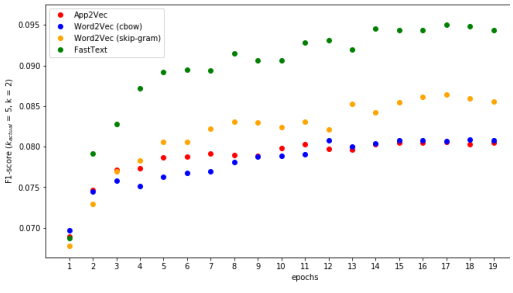

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 3, k = 10)
epochs

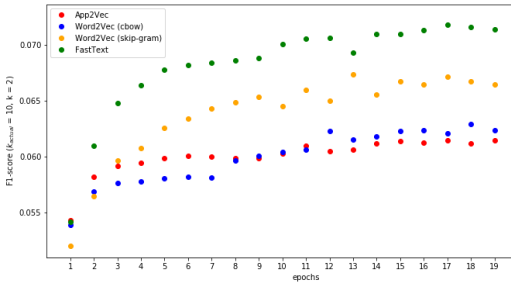

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 5, k = 10)
epochs

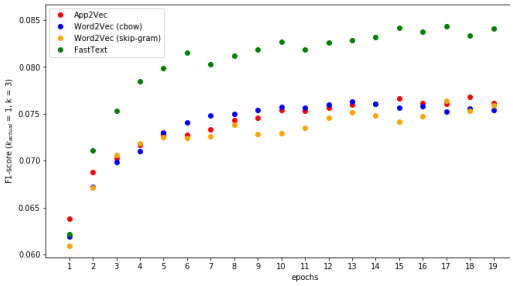

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
MAP@K (k_{actual} = 10, k = 10)
epochs

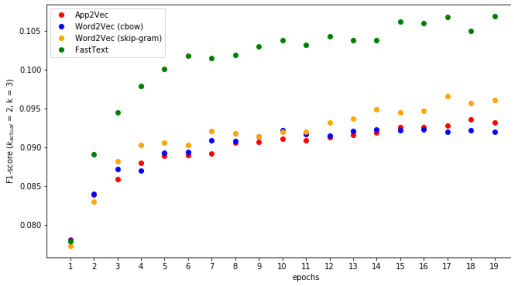
The following charts show the evolution of the *F1-score* metric during the training phase for the different candidate models and combinations of *k* (number of ordered similar sites considered by the model) and k_{actual} (number of similar sites retrieved from the *Alexa*'s service)

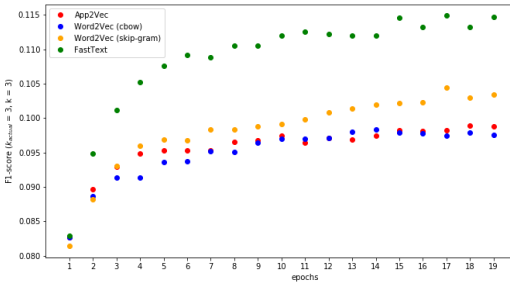


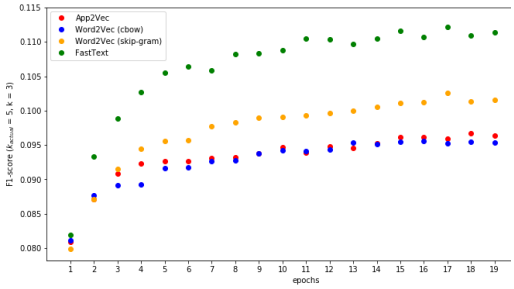

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 5, k = 2)
epochs

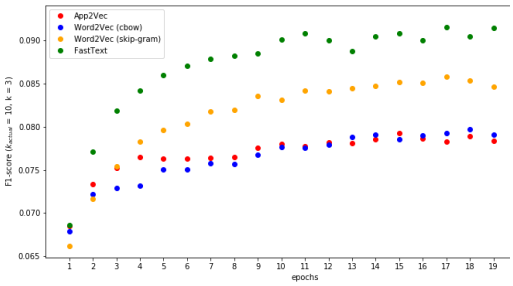

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 10, k = 2)
epochs

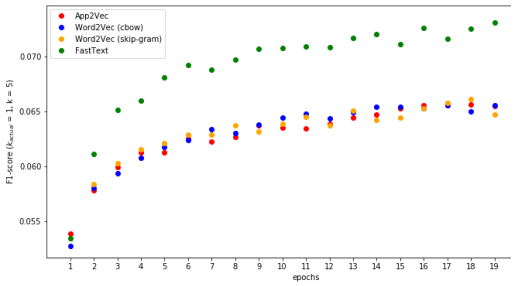

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 1, k = 3)
epochs

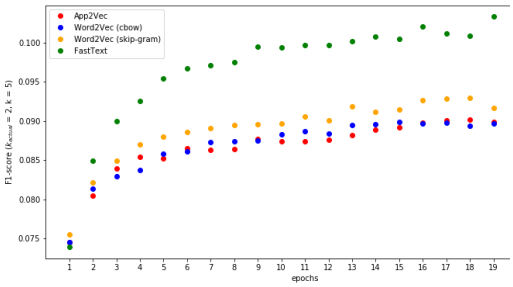

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 2, k = 3)
epochs

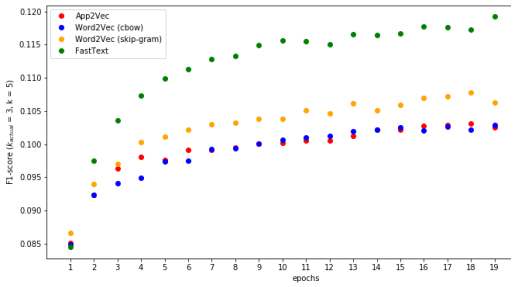

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 3, k = 3)
epochs

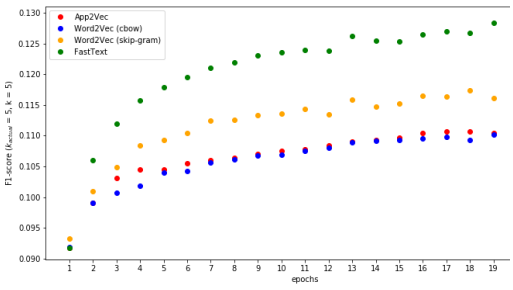

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 5, k = 3)
epochs

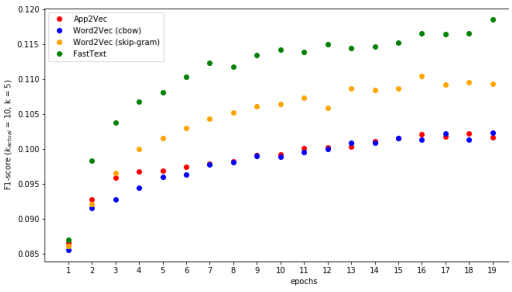

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 10, k = 3)
epochs

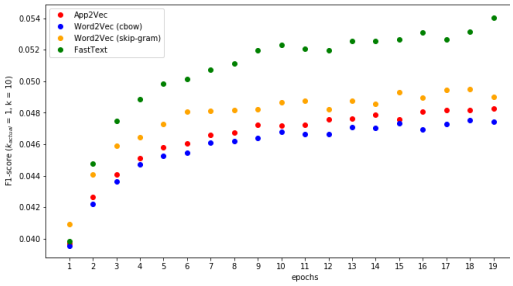

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 1, k = 5)
epochs

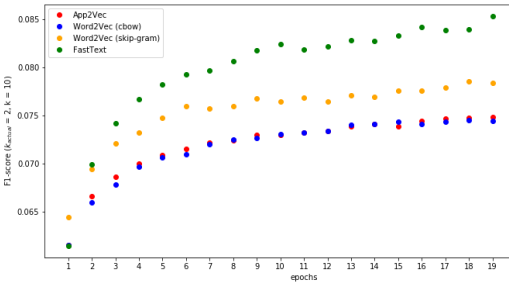

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 2, k = 5)
epochs

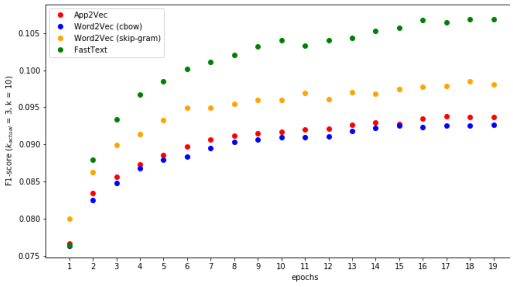

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 3, k = 5)
epochs

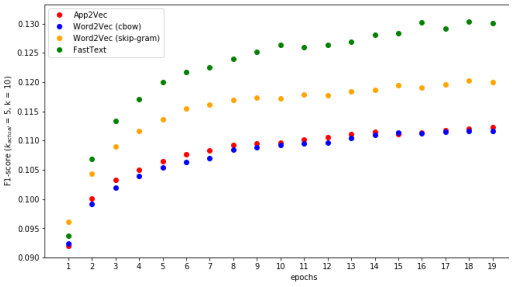

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 5, k = 5)
epochs

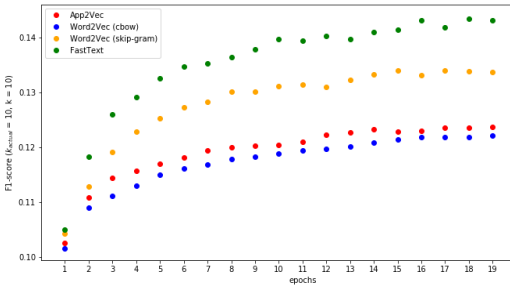

App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 10, k = 5)
epochs


App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 1, k = 10)
epochs


App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 2, k = 10)
epochs


App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 3, k = 10)
epochs


App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 5, k = 10)
epochs


App2Vec
Word2Vec (cbow)
Word2Vec (skip-gram)
FastText
F1-score (k_{actual} = 10, k = 10)
epochs