UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Selección óptima de rutinas de álgebra lineal dispersa usando aprendizaje automático

Informe de Proyecto de Grado presentado por

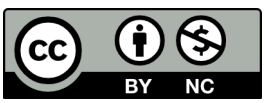
Mauricio Friss de Kereki

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Dr. Ing. Ernesto Dufrechou
Dr. Ing. Pablo Ezzatti

Montevideo, 4 de diciembre de 2025



Agradecimientos

Quiero agradecer a todos aquellos que me acompañaron durante la carrera y la elaboración de este trabajo. Principalmente quiero destacar a ellos:

A mi abuela Nubia, que no llego a verme entrar a la facultad pero estoy seguro que seguiría siendo mi fan número uno.

A mi pareja Abi, que tuve la hermosa suerte de tenerla siempre a mi lado durante esta etapa. Su amor, apoyo y compañía hicieron que pudiera llegar hasta este punto y ser quien soy hoy.

A mis tutores, Ernesto y Pablo, que me dieron la oportunidad de realizar el proyecto y explorar todas las ideas tangenciales que tuviera, brindándome seguridad y confianza en seguir adelante y terminarlo.

A mis padres, Mónica y Alejandro, que no solo colaboraron extensamente para corregir este trabajo, sino que desde muy chico me enseñaron lo que era la computación y fueron mi principal inspiración para entrar a esta carrera.

A Mica, Susana, Eugenio, y mis amigos Ximo y Bruno, que me acompañaron en los momentos más difíciles e hicieron que todos estos años fueran más llevaderos.

A todos aquellos compañeros que me ayudaron en todos los trabajos grupales a lo largo de la carrera.

Por último pero no menos importante, a la UdelaR y a todos los grandes docentes que hacen que sea una carrera tan desafiante como gratificante, preparándome perfectamente para el mundo laboral.

Resumen

La resolución de sistemas triangulares con matrices dispersas (SpTrSv, *Sparse Triangular Solve*) es fundamental en computación científica. Existen múltiples algoritmos que implementan esta operación para GPU con rendimientos muy variables según la estructura de cada matriz, sin que ninguno sea óptimo universalmente. En este trabajo se desarrollan las bases para un sistema de selección automática del mejor algoritmo mediante aprendizaje automático, que minimice el tiempo total de preprocesamiento, predicción y ejecución.

Se evaluaron 17 modelos de diferentes familias de aprendizaje automático: árboles de decisión (Random Forest, Extra Trees), boosting (Gradient Boosting, XGBoost, LightGBM, CatBoost), métodos lineales y redes neuronales. Los modelos se entrenaron con matrices de SuiteSparse usando Optuna para optimizar hiperparámetros. Para evaluar el rendimiento de los modelos se definió una nueva métrica, TAR, que mide el sobrecosto de ejecutar el algoritmo predicho sobre el óptimo.

A su vez, se analizaron las características medidas de las matrices, planteándose diferentes subconjuntos de las mismas con distintos balances entre expresividad y complejidad de cálculo. Se propusieron tres nuevas características: distribución por 32 bandas horizontales, índice de Gini y P-ratio. También se desarrolló una técnica de etiquetado que permite identificar las matrices que tienen un comportamiento similar frente a dos algoritmos, evitando penalizar predicciones entre algoritmos equivalentes.

Para expandir el conjunto de entrenamiento se implementó un sistema de generación de matrices sintéticas con Redes Generativas Adversarial. El proceso convierte matrices a imágenes de 128×128 pixeles donde cada píxel representa densidad de elementos distintos a cero, entrena una red para generar nuevas imágenes, y las expande a matrices dispersas. Se generaron 3500 matrices sintéticas validadas estadísticamente.

Los resultados superan ampliamente las líneas base. Para una ejecución de SpTrSv, las elecciones de algoritmos con Gradient Boosting agregaron solo un 4 % de diferencia con el tiempo óptimo, ampliamente mejor que el 872 % de sobrecosto esperado eligiendo aleatoriamente los algoritmos, con similares resultados en otros escenarios. La evaluación muestra que el enfoque puede generar importantes mejoras en casos de uso típicos, donde se resuelven decenas de sistemas con la misma matriz.

Palabras clave: Álgebra Lineal Numérica, SpTrSv, Aprendizaje Automático, Generación de Matrices Dispersas

Índice general

Capítulo 1

Introducción

La resolución de sistemas triangulares dispersos (SpTrSv) es una operación fundamental que aparece en forma recurrente en algoritmos de álgebra lineal numérica. Cuando se factoriza una matriz usando LU o Cholesky, cuando se aplican precondicionadores en métodos iterativos, o cuando se resuelven problemas de mínimos cuadrados, inevitablemente se termina resolviendo uno o varios sistemas triangulares [1]. En aplicaciones reales, esta operación puede ejecutarse cientos o miles de veces en un mismo proceso, por lo que optimizarla tiene un impacto directo en múltiples áreas de la computación científica.

Las unidades de procesamiento gráfico (GPU) cambiaron el panorama de la computación de alto rendimiento [2]. Mejoran mucho más rápido que los procesadores tradicionales, por lo que resultan más interesantes para desarrollar algoritmos de álgebra lineal, aunque estas mejoras solo se notan cuando se pueden aprovechar al máximo explotando sus capacidades de procesamiento paralelo. El problema específico de SpTrSv es complejo porque la naturaleza secuencial de resolver sistemas triangulares limita cuánto se puede paralelizar, y diferentes estructuras de matrices necesitan estrategias completamente distintas.

Como respuesta a estos problemas, se desarrollaron varios algoritmos para ejecutar SpTrSv en GPU. Algunos calculan las dependencias entre filas y planifican cómo repartir el trabajo antes de ejecutar y se sincronizan para avanzar a siguientes niveles, otros evitan las sincronizaciones esperando que aparezcan los resultados que necesitan, entre otras metodologías [3]. Cada uno tiene sus ventajas para ciertos tipos de matrices, pero también sus desventajas para otros. Además, muchos requieren una fase de análisis previa que puede ser costosa, algo que puede valer la pena solo si se ejecuta el algoritmo varias veces sobre matrices con el mismo patrón de dispersión.

1.1. Definición del Problema

El problema que aborda este trabajo es: dada una matriz triangular dispersa, seleccionar el algoritmo de SpTrSv que minimice el tiempo total de ejecución.

Este tiempo total incluye todo el proceso, desde medir características de la matriz, hacer la predicción del modelo, hasta la ejecución del algoritmo elegido, incluyendo cualquier preprocesamiento que este necesite.

Las características estructurales que determinan qué algoritmo será más eficiente no son claras. Una matriz puede tener pocas filas pero con muchas dependencias entre ellas, lo cual favorece ciertos algoritmos. Otra puede tener muchas filas pero con pocas dependencias cada una, favoreciendo otros algoritmos completamente distintos. La distribución espacial de los elementos no nulos, la profundidad del grafo de dependencias entre filas, qué tan desbalanceada está la carga de trabajo entre filas, y muchas otras propiedades impactan directamente en el rendimiento de los distintos algoritmos.

La elección óptima depende del contexto de uso. Si SpTrSv se ejecuta una sola vez, usar algoritmos sin preprocesamiento suele ser mejor porque no hay tiempo para amortizar. Pero si se ejecuta muchas veces sobre matrices similares, como puede suceder en métodos iterativos [1], entonces invertir tiempo en un análisis previo de las dependencias entre filas que mejore la asignación de trabajo a los procesadores durante la resolución puede resultar en menos tiempo total.

1.2. Objetivos

El objetivo general es desarrollar un sistema de selección automática de algoritmos de SpTrSv en *GPU* basado en aprendizaje automático que minimice el tiempo total de ejecución, incluyendo el preprocesamiento de la matriz, la predicción del modelo y la ejecución del algoritmo seleccionado.

Para alcanzar este objetivo, se establecen distintos objetivos intermedios.

Primero, evaluar un conjunto variado de modelos de aprendizaje automático para el problema de selección de algoritmos de SpTrSv, en escenarios diferentes.

Segundo, definir y validar nuevas características que capturen distintos aspectos estructurales de las matrices dispersas, relevantes para el rendimiento de algoritmos de SpTrSv, definiendo conjuntos de características con diferentes balances entre información y costo computacional.

Tercero, diseñar e implementar un sistema de generación de matrices sintéticas que replique las propiedades de matrices reales, validando estadísticamente que las distribuciones de características se mantienen consistentes.

Cuarto, evaluar el impacto del uso de las matrices sintéticas en la capacidad de generalización de los modelos, determinando si el aumento del conjunto de entrenamiento mejora las predicciones en matrices reales nuevas.

1.3. Organización del Documento

El resto de este documento se organiza en los siguientes capítulos.

El Capítulo 2 presenta los fundamentos teóricos necesarios para el trabajo y los antecedentes: el problema de SpTrSv, formatos de matrices dispersas y

algoritmos existentes, el uso de aprendizaje automático para selección de algoritmos en trabajos previos, bibliotecas con selección automática, y técnicas de generación de matrices sintéticas.

El Capítulo 3 describe la metodología del trabajo: definición de métricas de evaluación, los tipos de modelos utilizados, técnicas de combinación de algoritmos y optimización de hiperparámetros, descripción de características base y nuevas propuestas, y el proceso completo de generación de matrices sintéticas con Redes Generativas Adversarial (GANs).

El Capítulo 4 presenta los resultados experimentales: evaluación de varios modelos en distintos escenarios, impacto de la combinación de algoritmos, análisis de los conjuntos de características, validación de las matrices sintéticas, y resultados finales con el conjunto de entrenamiento expandido.

El Capítulo 5 evalúa el cumplimiento de objetivos y propone trabajo futuro.

Capítulo 2

Marco teórico y revisión de antecedentes

Este capítulo presenta los fundamentos teóricos y el estado del arte relacionados con este trabajo. Para entender por qué este problema es importante y cómo abordarlo, es necesario revisar cuatro áreas vinculadas. Primero, en la Sección 2.1 se presenta el problema de resolver sistemas triangulares dispersos. Se explica qué es esta operación, por qué es tan importante en álgebra lineal numérica, y qué hace tan difícil ejecutarla de forma eficiente en *GPUs*. También se revisan los distintos formatos para almacenar matrices dispersas y los algoritmos que se han desarrollado para trabajar con ellas.

En la Sección 2.2 se analiza cómo se ha usado el aprendizaje automático para elegir algoritmos en problemas similares, especialmente en la multiplicación matriz-vector (SpMV), que es la operación más estudiada del álgebra lineal numérica dispersa. Se identifican qué técnicas funcionan mejor y qué problemas aparecen al aplicar aprendizaje automático a estos problemas.

La Sección 2.3.1 revisa bibliotecas existentes que ya hacen selección automática de algoritmos o ejecución eficiente de operaciones sobre matrices dispersas.

Por último, la Sección 2.4 trata el problema de la escasez de matrices dispersas reales para entrenar modelos de aprendizaje automático. Se revisan técnicas para generar matrices sintéticas, especialmente usando GANs, que permiten crear nuevas matrices que se parecen a las reales y así expandir el conjunto de entrenamiento.

2.1. Álgebra Lineal Numérica

El álgebra lineal es una rama de las matemáticas que se enfoca en el estudio de matrices, vectores, espacios vectoriales y los algoritmos sobre estos. El álgebra lineal numérica es el estudio de la implementación de estos algoritmos en distintas computadoras y de los problemas que conllevan sus limitaciones.

Existen una gran variedad de áreas de estudio dentro del álgebra lineal alrededor de matrices, como el estudio de algoritmos para hallar los valores propios, o la solución de sistemas de ecuaciones lineales [1]. Un área particular, que es la que involucra este trabajo, es aquella que se enfoca en rediseñar aquellos algoritmos fundamentales sobre matrices, como la multiplicación de matriz por vector, matriz por matriz o la solución de sistemas triangulares sobre matrices, donde la relación de elementos distintos a cero sobre la cantidad de elementos es relativamente baja, conocidas como matrices dispersas.

Estudios en esta área permiten optimizar el tamaño en memoria y la cantidad de operaciones necesarias para trabajar con matrices que serian demasiado grandes para los algoritmos típicos. Este tipo de matrices surgen de modelar problemas de distintas áreas de la ciencia y la ingeniería, como los problemas de grafos no fuertemente conectados, o la resolución de ecuaciones diferenciales parciales [4].

2.1.1. Formatos de matrices dispersas

Como se planteó anteriormente, las matrices dispersas surgen de distintas áreas de ingeniería, por lo que inevitablemente, la distribución de los elementos también puede seguir infinitos patrones distintos. El objetivo es desperdiciar la menor cantidad de almacenamiento en ceros, por lo que es necesario almacenar los elementos distintos a ceros junto con la mínima información necesaria para poder reconstruir la matriz. Es por esto que existen muchos formatos de almacenamiento con el fin de optimizar aún más el uso de memoria y el posterior procesamiento explotando los patrones estructurales de las mismas.

Por ejemplo, un formato muy simple sería guardar ambas coordenadas de cada elemento distinto de cero y el elemento mismo. Este formato es conocido como *Coordinate* (COO), y se basa en almacenar 3 listas, una de los índices de fila, otra de los índices de columna y por último una de los elementos. Este funciona como base para otros formatos, como el formato *Compressed Sparse Row* (CSR), que comprime la lista de índices de fila, almacenando solo el índice donde comienza cada fila en las otras dos listas. De esta forma, si una matriz presenta n elementos en la misma fila, no hace falta guardar el indice de la fila n veces como en el formato COO. Esto resulta útil en matrices dispersas con filas con varios elementos cada una, pero matrices con pocas filas con más de un elemento es posible que ocupen más espacio en CSR que en COO, como se ve en la figura 2.1.

Los formatos en los que se almacenan las matrices no solo deben ser eficientes en términos de memoria, sino que están también fuertemente vinculados con el rendimiento al realizar las distintas operaciones. Cada formato implica la necesidad de un nuevo algoritmo para cada una de las operaciones que se quieran realizar con estas matrices. En operaciones sobre matrices dispersas, la relación dispar entre el costo de una operación de punto flotante y la transferencia de un número desde la memoria a un registro provoca que el principal cuello de botella de la mayoría de algoritmos sean los accesos a memoria, siendo necesario un patrón de accesos óptimo para no tener grandes penalidades. Es por esto

Figura 2.1: Matriz de ejemplo en formatos COO y CSR.

que los formatos intentan balancear la eficiencia en espacio de memoria con la velocidad en la que se pueden ejecutar estas operaciones.

Existen muchos trabajos donde se comparan formatos de forma empírica, como por ejemplo [5] donde se plantean distintos criterios para comparar y evaluar diferentes formatos, más alla de la comparación de eficiencia al realizar la multiplicación de una matriz por un vector.

2.1.2. Multiplicación matriz por vector y resolución de sistemas triangulares

Existe una gran variedad de operaciones que se pueden realizar con matrices dispersas, pero posiblemente la más estudiada es la multiplicación de una matriz por vector (conocido por sus siglas en ingles, SpMV) [6]. Esto se debe a que esta operación se encuentra en la base de muchos otros algoritmos y se suele realizar varias veces por algoritmo, por lo que cualquier optimización puede implicar grandes mejoras en problemas de muchas áreas de la ingeniería.

Esta operación es fuertemente paralelizable, dado que cada fila se puede procesar de forma independiente y también, dentro de cada fila solo se debe tener consideración al sumar todos los resultados intermedios, por lo que resulta muy atractivo hacer uso de la *GPU* para realizar esta tarea. Existen distintos algoritmos como aquellos presentados en [6] y [7] para realizar la multiplicación de forma paralela. Estos algoritmos deben considerar problemas como la distribución no uniforme de los datos, los accesos a memoria no continua y la sincronización entre procesadores trabajando sobre la misma fila. Estos problemas dependen totalmente del formato en el que se encuentra la matriz y estos problemas se pueden presentar a distintas medidas según la matriz, por lo

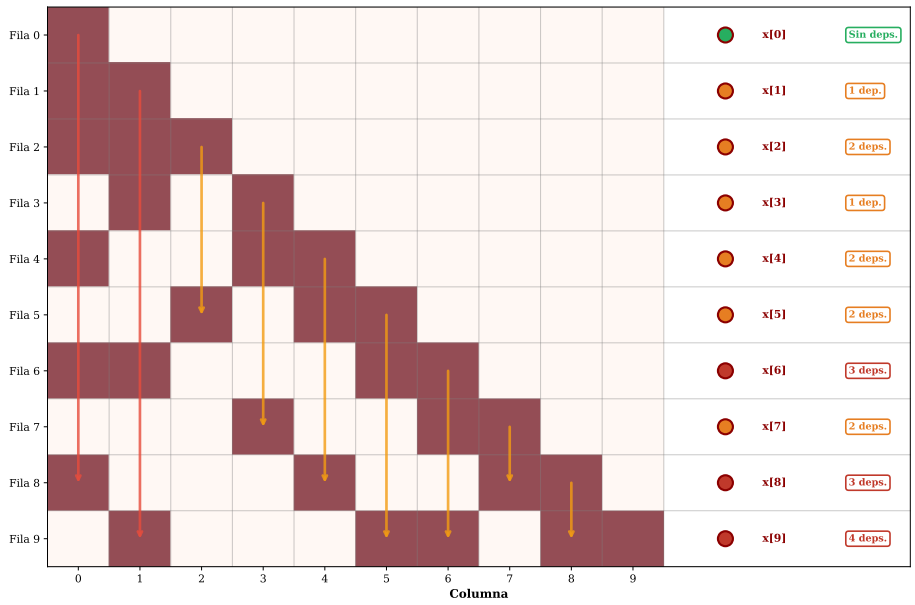


Figura 2.2: Matriz triangular de ejemplo donde se destacan las dependencias entre filas y la cantidad de filas de las que depende cada una.

que resulta imposible considerar un algoritmo como el definitivo, porque podría fácilmente ser el más veloz con una matriz pero a su vez ser el más lento con otra.

Otra operación muy estudiada es la de la resolución de sistemas triangulares (SpTrSv) por su gran utilidad y versatilidad, apareciendo en problemas como el de mínimos cuadrados, donde se realiza esta operación más de una vez, por lo que también optimizaciones en esta operación implican grandes mejoras en otros algoritmos más complejos.

Esta operación resulta mucho más compleja de paralelizar debido a dependencias entre filas para poder calcular cada resultado, por lo que para poder paralelizar, los algoritmos suelen comenzar con una fase de preprocesamiento muy fuerte, donde se decide el camino a seguir. A diferencia de SpMV, las principales optimizaciones no se basan en cambiar el formato en el que se guarda la matriz, sino en la configuración de los parámetros que deciden cómo se va a paralelizar y que métodos se van a seguir para cada fila de la matriz.

2.2. Aprendizaje Automático

El aprendizaje automático es un área dentro de la inteligencia artificial que se interesa en reconocer patrones y su importancia en datos de entrenamiento, con el fin de poder inferir información sobre nuevos datos sobre los que no se entrenó.

Esto puede implicar inferir categorías a las que pertenece el dato, valores que podría tomar en distintas categorías y más. Existen innumerables algoritmos con este fin, pero a grandes rasgos, se pueden dividir en dos categorías, en aprendizaje supervisado y en aprendizaje no supervisado.

La diferencia entre ellos radica en el conocimiento que se tiene sobre los datos de entrenamiento. En principio, en el aprendizaje no supervisado no es necesario saber como clasificarlos mientras que en el aprendizaje supervisado es necesario saber los resultados esperados de los modelos sobre el conjunto de entrenamiento. Esto no implica que si se conoce el resultado esperado, se deben utilizar algoritmos del aprendizaje automático supervisado, dado que puede ser igual de gran utilidad trabajar el resultado como una característica más.

Antes de evaluar cualquier modelo de aprendizaje automático, es necesario entender ciertos conceptos comunes a estos, que requieren atención para poder obtener buenos resultados al entrenar.

Datos de entrada

Todo algoritmo tiene ciertos valores que se ingresan para su procesamiento. Los algoritmos de aprendizaje automático no son la excepción y, en su mayoría, toman como entrada un conjunto de vectores, siendo cada vector el representante de un dato. Dependiendo de la naturaleza del problema a estudiar, se podría conocer (o calcular, como se presentará luego para utilizar en este trabajo) el resultado esperado que devuelva el algoritmo para los datos de entrenamiento.

A este conjunto de entrada se le llama el conjunto de entrenamiento y los modelos se ajustan para minimizar distintas funciones de pérdida, que son aquellas que determinan qué tan bien se ajustó el modelo a los datos. La función de perdida y la forma en la que se reajustan los valores internos de los modelos, son los principales diferenciales entre modelos.

Características

La mayoría de los datos sobre los que se trabaja resultan imposibles de ingresar completamente a los algoritmos, dado que el formato necesario para la entrada no es el mismo en el que se tienen los datos, y es por esto que se debe elegir un conjunto representativo del mismo. Para esto, se puede medir cada dato en n características e ingresar las medidas en un vector de n dimensiones.

La elección de las características es muy importante y debe realizarse con cuidado, dado que demasiadas características pueden ocasionar que el modelo no se ajuste correctamente al problema y no logre determinar correctamente las relaciones entre ellas, mientras que con demasiado poco, es muy posible que los vectores representantes no capturen la información importante del dato que representan y se pierda precisión.

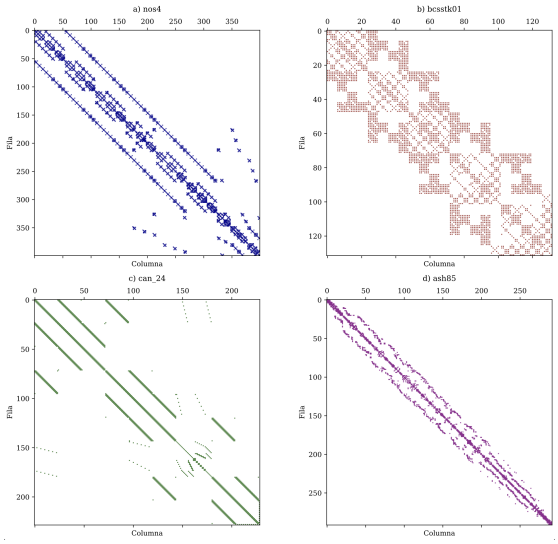


Figura 2.3: Matrices de ejemplo provenientes de [8] donde se puede ver como cuentan con distintos patrones y distintas distribuciones espaciales. Este es el tipo de información espacial que se busca capturar.

2.3. Predicción de desempeño mediante aprendizaje automático

Trabajos como [9] investigan el uso de distintos algoritmos de aprendizaje, de ambas categorías, con el fin de predecir cuál sería el mejor formato de representación de la matriz para ser utilizado en problemas de SpMV (multiplicación matriz - vector) en *GPUs*.

Los autores plantean utilizar algoritmos como XGBoost, presentado en [10]. A su vez, plantean un conjunto de características a considerar de las matrices como la cantidad de bloques de números contiguos por fila y el promedio del tamaño de los mismos, intentando capturar la mayor información posible sobre la distribución espacial de los elementos distintos a cero.

Por otro lado, trabajos como [11] utilizan aprendizaje automático para predecir los parámetros para un formato híbrido de representación de matrices. Este plantea una metodología interesante de elección de características de las matrices, planteando el uso de métodos para elegir las características más relevantes. El trabajo utiliza un método de selección de características conocido como Wrapper, presentado en [12], el cual se basa en ejecutar el algoritmo con subconjuntos de las características y así encontrar el subconjunto más efectivo.

Por otro lado, trabajos como [13] utilizan el aprendizaje profundo para realizar la tarea de predecir cuál formato y plataforma, *GPU* o *CPU*, es la más eficiente. El uso de aprendizaje profundo presenta una gran ventaja, en el sentido de que este tiene como entrada una matriz, no características representantes

de la matriz, y es este mismo que se encarga durante el entrenamiento de determinar las características relevantes a evaluar, en lugar de ser predefinidas por los investigadores.

En este trabajo los autores presentan un modelo de una red neuronal convolucional que tiene como entrada una imagen que tiene como objetivo representar la mayor cantidad de información de la matriz. Este proceso es luego refinado por los autores en [14] para que pueda representar aun más información, haciendo uso que los algoritmos de aprendizaje profundo son utilizados para el procesamiento de imágenes, por lo que la entrada es una matriz de píxeles con tres valores por celda, el RGB. Es en estos valores que codifican las verdaderas dimensiones de la matriz y la cantidad real de elementos distintos a cero, dado que la imagen representante que funciona de entrada es de tamaño 64×64.

También, en este trabajo se presenta una forma de generar matrices realistas para poder enriquecer el conjunto de entrenamiento, utilizando matrices reales para generar imágenes representantes y luego reescalandolas a distintas dimensiones.

Este último es luego profundizado en [15] haciendo que el modelo también considere las diferencias entre los tiempos de ejecución de los distintos algoritmos, para así minimizar el tiempo perdido con una elección errónea.

Por último, el trabajo [16] tiene un enfoque más global del proceso, considerando no solo la matriz sino que también un predictor que toma como entrada el programa que se va a ejecutar y este predice cuántas veces se va a ejecutar SpMV sobre la misma matriz, con el fin de considerar si las ejecuciones de la SpMV van a amortizar o no el costo de realizar la conversión a un formato distinto. Reportan, utilizando cuatro programas distintos de ejemplo, que este enfoque resulta con importantes mejoras de eficiencia globales (no solo menor tiempo de ejecución de SpMV).

Un detalle, es que considera solo un algoritmo por formato, brindados por CUSP [17], una librería actualizada por última vez en 2014. Trabajos como [18] se enfocan en minimizar el tiempo total del proceso, por lo que consideran el tiempo que toma evaluar las características y el tiempo de predicción de los modelos, presentando heurísticas para aproximar los valores de las características más complejas y reduciendo los modelos utilizados a solo aquellos que suelen ser veloces en predecir.

Todos estos trabajos presentan distintos abordajes al tema, con distintos algoritmos de aprendizaje automático, con variedad de resultados y efectividades. Muchos de estos compartían el enfoque de hallar el mejor clasificador para minimizar el tiempo de ejecución después de la predicción, pero este trabajo se va a enfocar en intentar obtener el tiempo total menor posible, es decir, considerando el tiempo de procesar la matriz para ser ingresada al modelo de aprendizaje automático, el tiempo que este demore en predecir, el tiempo para convertir a otra representación la matriz y/o transferir a la memoria de otro dispositivo si el predictor lo considera necesario y por último, la ejecución del algoritmo.

2.3.1. Bibliotecas

Existen trabajos como [19], que presentan bibliotecas encargadas de decidir automáticamente, por medio de heurísticas, la forma de representar la matriz y la implementación a utilizar, mientras que el usuario solo tiene que proporcionar los datos y elegir el problema a resolver, ya sea SpMV, SpTrSv o elevar una matriz a una potencia. Este trabajo presenta también un detalle interesante, y es la posibilidad de que el usuario brinde información sobre algunas características de la matriz, algo que en la práctica puede resultar útil, dado que se puede llegar a saber información general de las matrices que se van a procesar, y estos datos pueden servir para reducir el tiempo de procesamiento.

También en [20] se presenta una biblioteca que se encarga de elegir de forma eficiente no solo el algoritmo de SpMV, sino también los mejores parámetros para compilar el código. Para esta tarea, utilizan la herramienta Optuna [21], que se encarga de optimizar los hiperparámetros de los modelos (es decir, las distintas opciones que permiten configurarlos y manejan cómo se realiza el entrenamiento). Lo principal de este trabajo es que considera todas las partes del proceso, no solo la ejecución del algoritmo, dado que toma en cuenta la conversión desde CSR a cualquier otro formato y decide en base a si la ganancia cubre ese tiempo. Sin embargo, una limitación importante del trabajo es que las pruebas fueron realizadas solo en 30 matrices.

Por otro lado, en [22] se presenta una librería para optimizar SpTrSv. Para esto, considera la matriz y configura distintos parámetros de la ejecución del algoritmo SpTrSv obteniendo un tiempo de ejecución que en promedio es dos veces más veloz que ejecutar los algoritmos sin configuraciones ajustadas a la matriz. El trabajo considera el tiempo de preprocesamiento para poder ejecutar este algoritmo de forma eficiente, y plantea que suele ser unos ordenes de magnitud mayor que el tiempo de preprocesamiento de otros algoritmos, pero que aun así el tiempo global es comparable y prometedor.

2.4. Generación de Matrices Sintéticas

La generación de matrices sintéticas emergió como un área de investigación debido a las limitaciones en la disponibilidad de datos reales para el entrenamiento de modelos de aprendizaje automático. Esta problemática es especialmente relevante al trabajar con matrices dispersas, donde la diversidad y cantidad de matrices reales disponibles públicamente, como las de la colección SuiteSparse, es limitada y pueden resultar insuficientes para entrenar modelos que generalicen bien a nuevas matrices. Para superar este obstáculo, se han desarrollado diversas técnicas que van desde la modificación de matrices existentes hasta el uso de modelos generativos complejos.

2.4.1. Redes Generativas Adversarial (GANs)

Las Redes Generativas Adversarial, o GANs por sus siglas en inglés (*Generative Adversarial Networks*), son una clase de modelos de aprendizaje automático

presentada en el trabajo [23]. Su arquitectura se basa en un juego competitivo entre dos redes neuronales: el Generador y el Discriminador.

El objetivo del Generador es crear datos sintéticos que sean indistinguibles de los datos reales. Comienza con una entrada de ruido aleatorio (generalmente una muestra de un espacio latente, el cual es un espacio vectorial donde puntos cercanos representan cosas cercanas) y, a través de sus capas, transforma este ruido en una muestra de datos con la misma estructura que los datos de entrenamiento (por ejemplo, una imagen).

El Discriminador, por otro lado, actúa como un clasificador. Su tarea es determinar si una muestra de datos que recibe es real (proveniente del conjunto de datos de entrenamiento) o falsa (creada por el Generador).

El proceso de entrenamiento es un juego que se basa en el balance de las dos redes. El Generador intenta «engañar» al Discriminador produciendo datos cada vez más realistas, mientras que el Discriminador mejora continuamente su capacidad para diferenciar lo real de lo falso. Ambas redes se entrenan simultáneamente: el Generador se actualiza para minimizar la probabilidad de que el Discriminador detecte sus creaciones como falsas, y el Discriminador se actualiza para maximizar su precisión de clasificación. Este ciclo continúa durante varias generaciones hasta que el Generador produce datos tan convincentes que el Discriminador no puede distinguirlos de los reales con una precisión mayor al azar, alcanzando un punto de equilibrio.

Aunque su aplicación más conocida es la generación de imágenes fotorrealistas, la misma idea fundamental se ha explorado para generar otros tipos de datos estructurados, como matrices. Trabajos como [24] y [25] han adaptado esta arquitectura tratando las matrices como imágenes en escala de grises, donde el Generador aprende los patrones estructurales y de densidad para producir nuevas representaciones matriciales.

2.4.2. Generación por Perturbación de Matrices Reales

Paralelamente al desarrollo de modelos generativos profundos, existen otros enfoques para la creación de matrices sintéticas que se centran en replicar propiedades estructurales, estadísticas o numéricas específicas.

Un enfoque para expandir un conjunto de datos existente es la perturbación de matrices reales. Este método, utilizado en trabajos como [26], consiste en tomar una matriz real como base y aplicarle una serie de modificaciones controladas para crear una nueva sintética. Estas modificaciones pueden incluir:

- Añadir o eliminar elementos no nulos (nnz): Se pueden introducir o quitar elementos de forma aleatoria, posiblemente respetando la distribución de densidad original de la matriz.

- Intercambiar bloques o filas/columnas: Se pueden permutar secciones de la matriz para alterar su estructura sin cambiar las estadísticas locales.

El objetivo de estos métodos es generar nuevas matrices que, si bien son distintas a las originales, conservan las características estructurales a pequeña y gran escala (como la diagonalidad, el ancho de banda o la presencia de bloques densos) que son cruciales para el rendimiento de los algoritmos de álgebra lineal.

También, como se mencionó anteriormente, trabajos como [14] utilizan técnicas para reescalar las matrices reales a nuevos tamaños y entrenar con estas. Es debatible si estas matrices le presentan más información a los modelos o les repite la información que pueden obtener de las matrices originales.

2.4.3. Generación Basada en Propiedades Espectrales y Numéricas

Otra familia de métodos se enfoca en generar matrices que cumplan con propiedades matemáticas predefinidas, siendo de gran utilidad para la creación de bancos de pruebas para algoritmos numéricos.

El trabajo de [27] presenta un generador diseñado para crear matrices no hermitianas a gran escala con un espectro (conjunto de valores propios) específico. Esto es fundamental para analizar la convergencia de solucionadores iterativos, ya que su comportamiento depende directamente de la distribución de los valores propios de la matriz. El generador construye la matriz de manera que sus valores propios coincidan con los especificados por el usuario.

De manera similar, [28] se enfoca en la generación de matrices con valores singulares o números de condición controlados. El número de condición de una matriz mide su sensibilidad a errores en los datos de entrada; un número de condición alto indica que la matriz está «mal condicionada», es decir, numéricamente inestable para muchos algoritmos. La capacidad de generar matrices de gran tamaño con números de condición extremadamente altos o bajos es invaluable para realizar pruebas de estrés (*stress-testing*) y evaluar la robustez y precisión de los algoritmos numéricos en casos límite.

Estos enfoques, a diferencia de los métodos generativos similares a las GANs que aprenden patrones de forma implícita, construyen las matrices de forma explícita a partir de reglas y propiedades matemáticas, ofreciendo un control preciso sobre las características del conjunto de datos sintético resultante.

Capítulo 3

Metodología

Este capítulo presenta la metodología desarrollada para el problema de selección automática de algoritmos de SpTrSv mediante aprendizaje automático. El trabajo se enfoca en tres áreas principales: la evaluación de modelos de aprendizaje automático, la definición de características para representar matrices dispersas, y la generación de matrices sintéticas para expandir el conjunto de entrenamiento.

La Sección 3.1 define las métricas de evaluación utilizadas, compara el enfoque de clasificación y el de regresión, presenta la técnica de combinación de algoritmos, y describe el proceso de optimización de hiperparámetros mediante Optuna. Se evalúan 17 modelos diferentes en tres escenarios que representan distintas realidades de uso: una corrida, diez corridas y cien corridas del algoritmo sobre la misma matriz.

La Sección 3.2 presenta las características utilizadas para representar las matrices. Comienza con un conjunto base de nueve características de trabajos previos y se proponen tres nuevas: distribución por bandas horizontales, índice de Gini, y P-ratio. Estas se organizan en cuatro conjuntos que balancean información disponible contra tiempo de cálculo.

La Sección 3.3 describe el sistema de generación de matrices sintéticas usando GANs. Se detalla el proceso de downscale de matrices a imágenes de 128×128, el entrenamiento de la red generativa, y el upscale para crear nuevas matrices dispersas. Este proceso permite generar matrices adicionales para acompañar en el entrenamiento a las 1400 matrices reales disponibles en la colección SuiteSparse [8].

El objetivo en todos los casos es minimizar el tiempo total de ejecución, considerando el preprocesamiento de matrices y la ejecución del algoritmo seleccionado.

3.1. Evaluación de modelos

El primer objetivo es expandir el conjunto de modelos de aprendizaje automático evaluados en el trabajo [26] con distintas configuraciones de hiperparámetros. A su vez, interesa buscar metodologías de entrenamiento enfocadas en optimizar el tiempo de ejecución.

3.1.1. Definición de métricas

Antes de poder evaluar distintos modelos es necesario definir métricas para analizar su efectividad en el problema específico a resolver, la selección entre varias rutinas con el objetivo de minimizar el tiempo de ejecución. Un indicador general de efectividad de un modelo de clasificación es la precisión, definida como la cantidad de predicciones correctas sobre la cantidad de predicciones, considerando como correcta una predicción simplemente si el algoritmo escogido es el de menor tiempo de ejecución. Esta medida resulta simple de calcular, de interpretar y comparar. Sin embargo, resulta demasiado general, dado que en este caso, la elección incorrecta de un algoritmo no es grave por sí sola, sino que importa que al menos no agregue demasiado tiempo. Es decir, sería deseable un modelo que si no elige el mejor algoritmo, que al menos no elija de los peores.

Como segunda medida, se plantea el ***TAR***, tiempo agregado relativo. Esta es una medida personalizada, definida de la siguiente forma:

$$TAR \leftarrow promedio\left(\frac{TiempoPredicción - MejorTiempo}{MejorTiempo}\right) \tag{3.1}$$

Esta se interpreta como el sobrecosto promedio de ejecutar la predicción en lugar del mejor algoritmo. También se puede ver como la cantidad de veces extra que el mejor algoritmo se podría haber ejecutado, es decir, si esta medida da *2*, esto significa que el tiempo que toma el algoritmo predicho como mejor, resulta en un tiempo de ejecución equivalente a *3* veces el tiempo de ejecución del algoritmo que tomó menos tiempo.

Esta medida se considera más representativa de la efectividad de los modelos, dado que en la práctica no es tan grave elegir mal el algoritmo mientras su tiempo de ejecución no esté demasiado lejos del mejor. Por la naturaleza de los algoritmos, es común que en una misma matriz, varios se ejecuten en tiempos cercanos, por lo que no resulta adecuado penalizar al modelo de la misma forma si este predice un algoritmo incorrecto pero muy similar en tiempo, que si predice un algoritmo que ejecuta en un tiempo mucho mayor.

Se podría haber utilizado la métrica en una versión absoluta, realizando la suma de los tiempos agregados en vez del promedio, lo cuál privilegia los resultados para matrices grandes donde la diferencia de los algoritmos es mayor. Sin embargo, se decidió utilizar la métrica en su forma relativa dado que interesa mejorar para todas las matrices, no solo donde las diferencias entre los tiempos es mayor.

3.1.2. Clasificación vs Regresión

Con el fin de poder predecir cuál sería el mejor algoritmo a ejecutar, existen una infinidad de modelos compatibles con este propósito. Estos pueden categorizarse en dos grandes grupos según su salida, por un lado aquellos que retornan clasificaciones o etiquetas para una entrada desconocida (clasificadores), y por otro aquellos que retornan valores en un rango continuo (de regresión).

Modelos de regresión

Se plantearon dos enfoques generales distintos. El primero consistía en entrenar modelos de regresión utilizando los tiempos de ejecución de cada rutina de SpTrSv. Los modelos se entrenan para predecir los tiempos de ejecución, y posteriormente se elige la rutina que predijo con menor tiempo. Por un lado, este enfoque tiene la ventaja de considerar la realidad de forma más completa, dado que al utilizar los tiempos el modelo, en teoría, podría llegar a detectar mejor relaciones entre los algoritmos y eventualmente no solo mejorar la precisión, sino también el TAR. Un problema con este enfoque es el costo de la predicción, ya que requiere ejecutar un modelo distinto para predecir el tiempo de cada algoritmo, e iterar sobre los resultados para hallar el mejor. A su vez, los modelos de regresión suelen ser más lentos que los modelos clasificadores. Considerando esto, se deciden descartar los modelos de regresión para el resto del trabajo.

Modelos de clasificación

El otro enfoque se caracteriza en entrenar modelos clasificadores, etiquetando cada matriz de entrenamiento con el mejor algoritmo para resolverla. Esto implica la desventaja de que se pierde información dado que los modelos no saben cuanto tiempo tomó cada algoritmo, y lo que es más grave, le es imposible saber por cuánto tiempo fue mejor el mejor algoritmo contra el resto, por lo que no podría descubrir y hacer uso de patrones donde un par de algoritmos dan tiempos muy similares.

Como principal ventaja, este tipo de modelos suelen ser los más rápidos, dado que predecir una etiqueta es de las operaciones más básicas para los modelos estándar.

3.1.3. Combinación de algoritmos

Para solventar las desventajas mencionadas anteriormente, se propone una técnica de etiquetado de matrices, que llamaremos «combinar algoritmos». Esta consiste en agregar etiquetas especiales para los casos en que el segundo mejor algoritmo tiene un tiempo cercano al primero. Es decir, si el tiempo del segundo mejor algoritmo se encuentra dentro de un cierto margen de distancia del tiempo del mejor algoritmo, se etiqueta esa matriz como que ese par de algoritmos es el mejor. Todas las matrices donde ese mismo par son los dos mejores, serán etiquetadas de la misma forma.

Esto se realiza con el fin de que matrices donde las diferencias de tiempos son mínimas, no terminen siendo interpretadas por el modelo como totalmente distintas por el hecho de que en una fue mínimamente mejor uno y en otra, el otro algoritmo.

En teoría, este acercamiento podría dar mejores resultados, dado que lo que se intenta es obtener un modelo que determine a qué clase de matrices pertenece una matriz desconocida, y es posible que sean clases distintas aquellas donde da relativamente lo mismo elegir entre un algoritmo u otro y las matrices con un algoritmo claramente más óptimo. Es importante destacar que muchos de los modelos de aprendizaje automático escalan lineal o hasta cuadráticamente su complejidad según la cantidad de etiquetas o clases para clasificar [29], por lo que esto se debe considerar, dado que una mayor complejidad impacta directamente en el tiempo de predicción.

Una forma que podría resultar clara para hacer que los modelos conozcan los tiempos de los algoritmos y poder hacer uso de relaciones entre ellos sin tener que aceptar las desventajas de los modelos de regresión podría ser ingresar los tiempos como características de la matriz, pero esto tiene un claro problema, y es que la intención de obtener un modelo predictor es que este pueda enfrentarse a una matriz completamente desconocida y poder predecir cuál algoritmo usar, pero si un modelo es entrenado con unas características, estas mismas deben ser medidas en las matrices que se quieren evaluar, por lo que perdería el sentido si se necesita ejecutar todos los algoritmos para conseguir los tiempos.

3.1.4. Elección de hiperparámetros

Todos los modelos de aprendizaje automático dependen de una gran variedad de parámetros que determinan su comportamiento (conocidos como hiperparámetros) y por ende, su efectividad para aprender sobre el conjunto de datos. Se podría experimentar con distintos valores pero existen librerías que se pueden utilizar con este mismo propósito de una forma más eficiente.

Librerías como Optuna permiten definir una función, unos parámetros y los valores que pueden tomar los mismos y un objetivo (maximizar o minimizar el valor retornado por la función) y este se encarga de intentar hacer una búsqueda eficiente sobre el conjunto de combinaciones de valores para hallar la mejor combinación en una cantidad finita de pasos.

Por medio de esta biblioteca se puede entonces ejecutar un modelo con unos hiperparámetros, evaluar su rendimiento en términos de precisión o TAR, y que esta se encargue de encontrar los hiperparámetros que maximizan la precisión, minimizan el TAR o ambas a la vez.

Esto permite que aunque el modelo desconozca los tiempos relativos o cuanto tiempo agrega el algoritmo que predijo, se obtenga el modelo que de alguna forma, alcanzando el mejor rendimiento según esta métrica.

3.1.5. Funciones de pérdida

Muchos modelos de aprendizaje automático se basan en optimizar para minimizar una cierta función de pérdida, la cual evalúa el rendimiento y decide cómo ajustar sus parámetros internos [30]. La mayoría de los modelos usan alguna función de pérdida o permiten elegir entre varias predefinidas, porque normalmente el paso de ajuste de parámetros depende de conocer las derivadas de la función de pérdida para decidir los saltos y las direcciones, pero algunos permiten modificar la función de pérdida por una totalmente personalizada.

En teoría, con modelos que permiten utilizar funciones personalizadas, se podría usar el TAR como función de pérdida y que el modelo trabaje en optimizarla. Esto permitiría entrenar utilizando la mayor cantidad de información posible y, después, no depender de los datos de tiempo para realizar las predicciones, dado que la función de pérdida solo es necesaria para reajustar los parámetros internos, no para predecir.

Otro punto a considerar, es que el objetivo final es poder crear una biblioteca a la cual el usuario solo deba proporcionarle la matriz dispersa y los vectores sobre los cuales desea ejecutar el sistema triangular de ecuaciones y esta misma se encargue de elegir cuál sería el método más veloz, ejecutarlo y proporcionarle los resultados al usuario, intentando al menos garantizar un tiempo total de ejecución más rápido a que siempre se use el mismo algoritmo o elegir al azar entre los posibles.

Para poder tener disponibles una mayor cantidad de herramientas para el aprendizaje automático y por sus facilidades a la hora de programar, para el procesamiento de los datos y el entrenamiento de los modelos se utiliza Python, mientras que los algoritmos de SpTrSv se encuentran programados en CUDA por lo que para poder hacer que la biblioteca sea lo más eficiente posible, interesa que se puedan exportar los modelos entrenados a C++ y poder trabajar en el mismo programa con los algoritmos en CUDA.

Se evaluaron 17 algoritmos de aprendizaje automático de diferentes familias:

- Métodos basados en árboles: Random Forest, Extra Trees, Arboles de decisión, XGBoost y XGBoost con random forests [10].
- Métodos de boosting: Gradient Boosting, AdaBoost [31], LightGBM [32], CatBoost [33]
- Métodos lineales: Ridge, SGD, Passive Aggressive
- Otros: SVM, KNN, Naive Bayes, MLP

Estos fueron seleccionados para cubrir una gran variedad de modelos, intentando tener de los principales métodos de aprendizaje automático.

3.2. Características

Todo modelo de aprendizaje automático supervisado requiere de entradas de formato consistente y su correspondiente solución, ya sea una etiqueta para

los modelos de clasificación o un número para los modelos de regresión. En problemas clásicos, puede resultar claro cuál sería la entrada a los modelos, por ejemplo, si se quiere generar un modelo que sugiere jugadas de ajedrez, se podría ingresar el estado actual del tablero, si se quieren clasificar razas de perros se podría usar como entrada distintas medidas de las partes del cuerpo, el color, cantidad y forma de los dientes, o hasta una foto del perro.

En este caso, se desea clasificar matrices dispersas de tamaños indeterminados, por lo que resulta imposible usarlas como entrada directa. Al igual que en el caso de clasificar razas de perro, se deben elegir características que en conjunto logren describir la matriz original, o al menos, los datos necesarios para determinar cuál algoritmo será el más eficiente para esa matriz.

Mientras que en el caso de clasificar perros resultan obvias las características cuantificables de un perro, cuando se trabaja con matrices no es tan claro y existen incontables propiedades posibles que se podrían medir. La elección de características impactan muy fuertemente en los resultados de los modelos, dado que estas deben capturar los aspectos estructurales de la matriz que más influyen en el rendimiento relativo de los distintos algoritmos de SpTrSv.

3.2.1. Características Base

En el trabajo [26] se define un conjunto inicial de 9 características, que se tomaron como punto de partida en este trabajo. Estas características se pueden categorizar según el aspecto estructural que intentan capturar:

Características dimensionales básicas:

- n: Tamaño de la matriz (número de filas/columnas)
- nnz: Cantidad total de elementos no nulos

Características de distribución por filas:

- *max nnz*: Cantidad máxima de elementos no nulos en una fila
- *nnz promedio por fila*: Media de elementos no nulos por fila
- *desviación estándar de nnz*: Variabilidad en la distribución de elementos por fila

Características de estructura y localidad:

- *bandwidth promedio*: Distancia promedio del elemento no nulo más lejano a la diagonal principal
- *n levels*: Profundidad del grafo de dependencias entre las filas

Características de paralelización específicas:

- *locality simple*: Medición de paralelismo potencial para el algoritmo simple
- *locality multi*: Medición de paralelismo potencial para el algoritmo multi-row

Estas características demostraron ser relevantes en trabajos como [26], pero pueden no ser suficientes para capturar completamente los patrones estructurales de la matriz, como la ubicación espacial de los elementos no nulos.

3.2.2. Características Propuestas

Con el objetivo de enriquecer la representación de las matrices y potencialmente mejorar la eficiencia de los modelos, se desarrollaron tres características nuevas. Se buscó capturar información sobre la disposición de los elementos no nulos en la matriz. Por ejemplo, si existieran zonas o filas mucho más densas que el promedio, estas pueden complicar el balance de carga e impactar directamente en el rendimiento de ciertos algoritmos

Distribución por Bandas Horizontales

Se implementó una caracterización de la matriz dividiéndola en 32 bandas horizontales de igual tamaño. Para cada banda i, se calcula:

$$banda_i = \frac{nnz\_en\_banda_i}{nnz\_total} \tag{3.2}$$

Esta representación captura patrones de distribución espacial de los elementos no nulos que podrían influir en el rendimiento de algoritmos que procesan la matriz por bloques o que tienen sensibilidad a la localidad de los datos. Se eligieron 32 bandas porque se consideró que son una cantidad suficiente para detallar la distribución sin agregar demasiadas dimensiones a los modelos.

Índice de Gini

Se calculó el índice de Gini [34] para medir la desigualdad en la distribución de elementos no nulos, tanto por filas como por las bandas horizontales definidas anteriormente:

$$Gini = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} |x_i - x_j|}{2n \sum_{i=1}^{n} x_i} \tag{3.3}$$

donde x_i representa la cantidad de elementos no nulos en la fila i (o banda i en el segundo caso).

El índice de Gini por filas (*gini_filas*) mide qué tan desigual es la distribución de trabajo entre las filas, lo cual puede influir significativamente en algoritmos que paralelicen por filas. El índice de Gini por bandas (*gini_bandas*) captura la desigualdad en la distribución vertical de los elementos, complementando la información de las 32 bandas individuales.

P-Ratio

Se implementó la métrica P-ratio, que mide la concentración de elementos. Esta métrica se interpreta como el porcentaje p de filas (o bandas) que contiene

el $(1-p)\,\%$ de los elementos no nulos. Específicamente, si el P-ratio es 20, significa que el 20 % de las filas más densas contiene el 80 % de los elementos no nulos.

Se calcularon dos variantes:

- *p_ratio_filas*: Concentración de elementos por filas
- *p_ratio_bandas*: Concentración de elementos por bandas horizontales

Al igual que el indice de Gini, esta métrica es particularmente relevante para algoritmos capaces de identificar y procesar de manera diferente las regiones más densas de la matriz.

3.2.3. Conjuntos de Características

Siempre es posible encontrar características cada vez más complejas, pero que se vinculen más con los tiempos de ejecución (por ejemplo, se podrían ejecutar los algoritmos con ciertas secciones de la matriz). Pero claramente el cálculo de las mismas tomaría un tiempo considerable que hasta podría hacer inútil al proceso de predicción porque si el procesamiento para predecir toma más tiempo que ejecutar cualquier algoritmo, no aporta ninguna ventaja para predecir.

Para evaluar el impacto de las diferentes características y considerar las restricciones prácticas de tiempo de cómputo, es decir, evaluar si con características más rápidas de calcular, se obtienen resultados suficientemente útiles, se definieron cuatro conjuntos de características:

Conjunto Original: Las 9 características presentadas en [26].
Conjunto Rápido Original: Únicamente las características n y nnz, que se pueden obtener en tiempo constante al acceder a los metadatos de la matriz dispersa almacenada.
Conjunto Completo: Todas las 45 características (9 originales + 32 bandas + 2 índices de Gini + 2 P-ratios), maximizando la información disponible para el modelo.
Conjunto Rápido Extendido: Las 2 características rápidas originales más las 32 medidas de bandas horizontales, totalizando 34 características.

La diferencia entre características «rápidas» y «complejas» es muy importante desde el punto de vista práctico. Las características rápidas (n, nnz, y las bandas horizontales) pueden calcularse en tiempo constante ya que implican una cantidad fija de operaciones simples sobre la estructura de datos de la matriz en CSR. En contraste, las características complejas requieren recorridos secuenciales, cálculos de dependencias, o análisis de grafos que escalan con el tamaño y la estructura de la matriz.

Esta categorización permite evaluar si el tiempo adicional requerido para calcular características más sofisticadas se justifica por una mejora significativa en la precisión de predicción, o si un conjunto más limitado pero rápido de calcular puede proporcionar resultados comparables.

3.3. Generación de Matrices Sintéticas

Durante el desarrollo del proyecto se detectó una limitación fundamental: la cantidad de matrices reales disponibles para entrenamiento es relativamente reducida. Se obtuvieron todas las matrices cuadradas de SuiteSparse y se eliminaron duplicadas según los nombres (para una misma matriz al descargarla de la colección se suelen obtener varias versiones de la misma, apenas cambiados los valores o la distribución).

Al final de este proceso, quedaron 1400 matrices que puede resultar insuficiente para el entrenamiento de modelos de aprendizaje automático, especialmente considerando la diversidad de estructuras de matrices dispersas que pueden existir en aplicaciones reales.

Para abordar esta limitación, se desarrolló un sistema de generación de matrices sintéticas que replica las características estructurales relevantes de las matrices reales, permitiendo generar conjuntos de entrenamiento de mayor tamaño y posiblemente más diversos.

El desafío principal en la generación de matrices sintéticas es capturar y reproducir los patrones estructurales complejos que caracterizan a las matrices dispersas provenientes de problemas reales. No resulta útil generar matrices totalmente aleatorias dado que no representan las posibles estructuras presentadas en los problemas reales, pero tampoco resulta conveniente sesgar el conjunto de datos sobrerrepresentando las estructuras ya presentes en los datos dado que esto no escalaría a nuevas matrices.

Se planteó una aproximación basada en representación visual de las matrices, tratándolas como imágenes donde la intensidad de cada píxel representa la densidad de elementos no nulos. Existen varios trabajos en los que se utilizan estas representaciones para agrandarlas a distintos tamaños y así obtener matrices nuevas. La novedad se encuentra en que esta representación también permite aplicar técnicas de generación de imágenes sintéticas, específicamente Redes Generativas Adversarial (GANs), para crear nuevas imágenes y expandirlas a nuevas matrices.

3.3.1. Proceso de Representación Visual

Downscale a representación de tamaño 128x128

Para cada matriz real de tamaño $n \times n$, se aplica un proceso de downscale que la mapea a una imagen de 128×128 píxeles. Este proceso divide conceptualmente la matriz original en una grilla uniforme de 128×128 regiones cuadradas.

Para una matriz de tamaño $n \times n$, cada región (i, j) en la grilla 128×128

corresponde a la región rectangular de la matriz original definida por:

$$fila_{inicio} = \lfloor \frac{i \cdot n}{128} \rfloor \tag{3.4}$$

$$fila_{fin} = \lfloor \frac{(i+1) \cdot n}{128} \rfloor \tag{3.5}$$

$$col_{inicio} = \lfloor \frac{j \cdot n}{128} \rfloor \tag{3.6}$$

$$col_{fin} = \lfloor \frac{(j+1) \cdot n}{128} \rfloor \tag{3.7}$$

Cálculo de Densidad Local

Para cada región, se calcula la densidad de elementos no nulos:

$$densidad_{i,j} = \frac{nnz\_en\_region_{i,j}}{celdas\_totales\_en\_region_{i,j}} \tag{3.8}$$

donde $nnz\_en\_region_{i,j}$ es la cantidad de elementos no nulos en la región correspondiente a la región (i, j), y $celdas\_totales\_en\_region_{i,j}$ es el número total de elementos de la matriz original en esa misma región.

Conversión a Escala de Grises

Los valores de densidad, que están en el rango $[0, 1]$, se convierten a enteros en el rango $[0, 255]$ para su representación como imágenes en escala de grises:

$$pixel_{i,j} = \lfloor densidad_{i,j} \times 255 \rfloor \tag{3.9}$$

Esta representación mantiene la información estructural esencial de la matriz original mientras la reduce a un formato manejable y uniforme para el entrenamiento de la GAN, dado que se mantiene constante para cualquier tamaño original de matriz. A su vez, al estar acotado a ciertos valores (de 0 a 255) la GAN converge de forma más rápida porque se reduce el espacio de valores posibles.

3.3.2. Entrenamiento de la Red Generativa Adversarial

Con el conjunto de imágenes de 128×128 pixeles generadas a partir de las 1400 matrices reales, se entrenó una GAN especializada en generar imágenes sintéticas en escala de grises que repliquen los patrones estructurales observados en las matrices reales.

Existen infinidad de redes posibles para implementar el Generador y el Discriminador de la GAN, pero las que suelen converger de forma más rápida en casos donde se cuenta con pocos datos son las DCGAN [23], las cuales usan redes profundas convolucionales para ambas redes. Como se mencionó, el Generador aprende a convertir ruido aleatorio en imágenes que logran confundir

al Discriminador, pero si se conocen patrones de las imágenes que se quieren generar, se puede usar estos patrones para el ruido y converger más rápido.

En este caso, se sabe que la gran mayoría de pixeles representan zonas vacías o muy cerca a ser vacías, por lo que ruido con números muy cercanos a 0, sera más cercano a las imágenes reales.

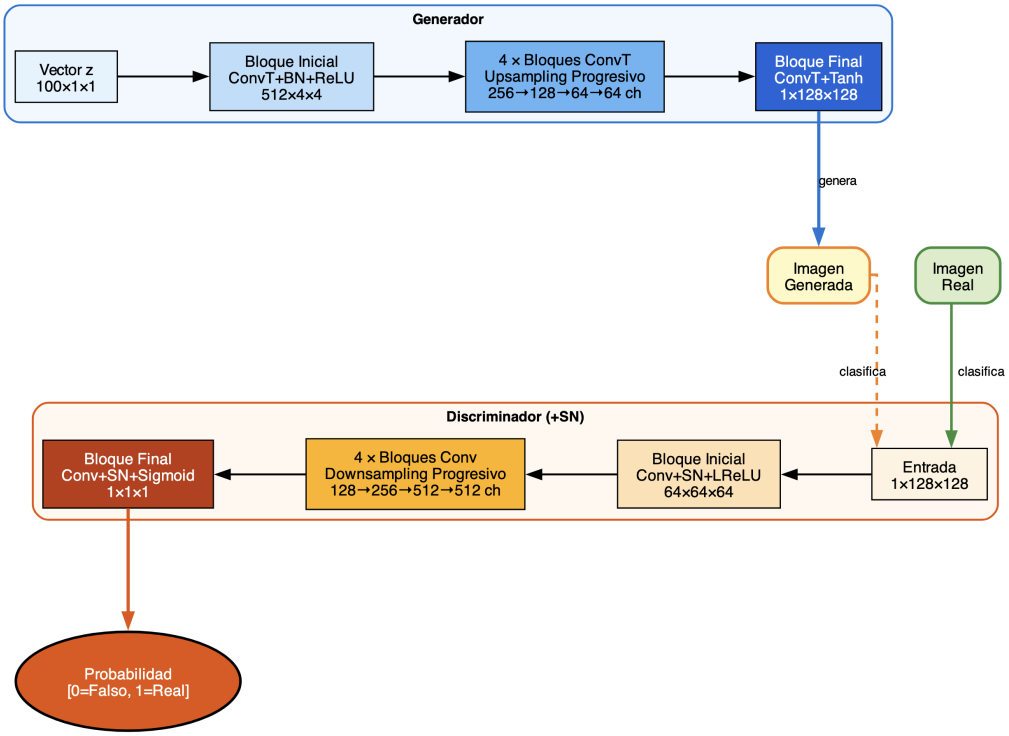


Figura 3.1: Arquitectura de la GAN. El **Generador** (arriba, azul) transforma un vector en una imagen sintética de 128×128 píxeles. El **Discriminador** (abajo, naranja) recibe imágenes reales y generadas y las clasifica, terminando en una probabilidad. Las capas intermedias repetitivas de cada red se agruparon para mayor claridad. Todos los bloques convolucionales del discriminador utilizan Normalización Espectral (SN).

Se diseñó la GAN repitiendo capas de convolución, normalización y funciones de activación ReLU para ambas redes siguiendo patrones estandares de la literatura para la creación de DCGANs [35]. A su vez, se envuelve todas las capas del Discriminador con normalización espectral, para evitar un problema común en las GAN, donde el Discriminador se vuelve demasiado fuerte y no le transmite información útil al Generador.

3.3.3. Proceso de Upscale e Inversión

Para convertir las imágenes sintéticas generadas por la GAN en matrices dispersas utilizables, se desarrolló un proceso de upscale que invierte la transformación original.

Definición del Tamaño Objetivo

Para definir el tamaño final de las matrices se implementó un generador de tamaños aleatorios basado en la distribución observada en las matrices reales. Utilizando SciPy con sus herramientas de análisis estadístico sobre la distribución de los tamaños de las matrices de SuiteSparse, se pudo ajustar una distribución logarítmica normal que replica la distribución de los tamaños reales:

$$n_{objetivo} = \lfloor e^{\mathcal{N}(\mu_{log}, \sigma^2_{log})} \rfloor \tag{3.10}$$

donde μ_{log} y σ_{log} son los parámetros de la distribución logarítmica ajustada a los datos reales.

Mapeo a Matriz Dispersa

Para una imagen sintética de 128×128 y un tamaño objetivo $n_{objetivo} \times n_{objetivo}$, cada píxel de la imagen se mapea a una región cuadrada de la matriz objetivo. La densidad representada por cada píxel determina probabilísticamente la cantidad de elementos no nulos a colocar en la región correspondiente.

Para cada píxel (i, j) con valor $pixel_{i,j}$:

$$densidad_{pixel} = \frac{pixel_{i,j}}{255} \tag{3.11}$$

$$nnz_{region} = \text{round}(densidad_{pixel} \times celdas\_en\_region) \tag{3.12}$$

Los elementos no nulos se distribuyen aleatoriamente dentro de cada región, manteniendo la densidad especificada por la imagen sintética.

3.3.4. Sistema de Generación Automatizado

Se implementó un sistema automatizado que ejecuta el ciclo presentado en el algoritmo 1.

Este sistema permite generar conjuntos de datos de entrenamiento de tamaño arbitrario, complementando las matrices reales con matrices sintéticas que preservan las características estructurales relevantes para el problema de selección de algoritmos de SpTrSv.

3.3.5. Validación de la Generación Sintética

Para asegurar que las matrices sintéticas generadas mantienen relevancia para el problema original, se implementó un proceso de validación que compara las distribuciones de características entre matrices reales y sintéticas, verificando que:

Algoritmo 1 Generación Automatizada de Datos Sintéticos

```
while true do
    n ← generar_tamaño_aleatorio()
    imagen ← generar_con_GAN()
    matriz ← upscale_matriz(imagen, n)
    guardar_matriz_en_disco(matriz)
    caracteristicas ← medir_características(matriz)
    tiempos ← ejecutar_algoritmos_SpTrSv(matriz)
    guardar_resultados(caracteristicas, tiempos)
    eliminar_matriz_del_disco(matriz)
end while
```

- Visualmente las matrices generadas se asemejan a las reales
- Las distribuciones de las características son estadísticamente similares
- Las distribuciones de los mejores algoritmos se mantienen similares

Para validar visualmente las matrices generadas se sigue el entrenamiento de la GAN a través de las generaciones y se controla que tiendan a parecerse a las matrices reales, terminando el entrenamiento cuando el progreso se detiene, es decir, cuando las matrices generadas se mantienen iguales.

A su vez, se validan sus características comparando aleatoriamente una cantidad de matrices reales y generadas. También, se evalúan los conjuntos totales y se comparan las medias de las estadísticas con el fin de detectar estadísticas donde difieren los conjuntos.

Capítulo 4

Evaluación Experimental

En el capítulo anterior se presentaron las tres líneas principales de trabajo de este proyecto: la evaluación de diferentes modelos de aprendizaje automático con sus respectivas configuraciones, la definición y medición de nuevas características para representar las matrices dispersas, y el desarrollo de un sistema de generación de matrices sintéticas basado en redes generativas adversarias (GANs). Cada una de estas componentes fue diseñada con el objetivo de mejorar la capacidad de predecir qué algoritmo de SpTrSv resultará más eficiente para una matriz dada.

Este capítulo presenta los resultados experimentales obtenidos al evaluar estas propuestas. La evaluación se organiza en cuatro secciones principales que abordan cada aspecto del trabajo de forma progresiva, utilizando los resultados previos para incorporar cada mejora propuesta.

En la primera sección se evalúan los 17 modelos de aprendizaje automático seleccionados, entrenados sobre el conjunto base de 1400 matrices reales de SuiteSparse. Para cada modelo se utiliza Optuna para encontrar la mejor configuración de hiperparámetros, y se reportan la precisión y el TAR en los tres escenarios definidos: una corrida, diez corridas y cien corridas del algoritmo sobre matrices similares. Estos escenarios permiten evaluar cómo cambia el comportamiento óptimo cuando se puede amortizar el costo del análisis previo que requieren algunos algoritmos.

La segunda sección incorpora la técnica de combinación de algoritmos presentada en el capítulo anterior. Cuando dos algoritmos tienen tiempos de ejecución muy similares en una matriz, se evalúa si etiquetarlos de forma especial durante el entrenamiento permite que los modelos aprendan a identificar estos casos ambiguos de forma más efectiva, potencialmente reduciendo el TAR y la precisión al no penalizar tanto los errores entre algoritmos equivalentes.

La tercera sección analiza el impacto de los diferentes conjuntos de características definidos. Se comparan los resultados obtenidos con el conjunto original de 9 características, el conjunto rápido de solo 2 características calculables en tiempo constante, el conjunto completo de 46 características que incluye todas las propuestas nuevas, y el conjunto rápido extendido que combina velo-

cidad de cálculo con mayor información mediante las 32 bandas horizontales. Esta comparación permite evaluar si las nuevas características aportan información relevante y si existe un balance entre precisión de predicción y tiempo de preprocesamiento.

Finalmente, la cuarta sección presenta los resultados del sistema de generación de matrices sintéticas. Primero se valida que el proceso de downscale-upscale preserve la información estructural relevante, luego se evalúa visualmente el progreso del entrenamiento de la GAN, y se comparan las distribuciones estadísticas de las matrices generadas contra las reales. La sección finaliza presentando los resultados obtenidos al entrenar los modelos con el conjunto expandido que incluye 3500 matrices sintéticas adicionales, evaluando si este aumento en la cantidad de datos de entrenamiento se acompaña de mejoras en la efectividad de los modelos.

A lo largo de todo el capítulo, los resultados se comparan contra el TAR esperado de elegir algoritmos al azar y el TAR de siempre elegir el mismo algoritmo. Los mejores resultados en cada escenario se destacan para facilitar la identificación de las configuraciones más prometedoras.

4.1. Modelos

Cómo se mencionó anteriormente, interesa la posibilidad de integrar el modelo de aprendizaje automático con una biblioteca en C++. Es por esto que se limitaron los modelos a los provenientes de cuatro librerías que presentan facilidades para ser exportados a otros lenguajes. De estas, se eligieron 17 algoritmos distintos que cubren una gran variedad de familias de modelos.

Los modelos, presentados en la tabla 4.1, son entrenados sobre el mismo conjunto de entrenamiento.

El conjunto total de entrenamiento se encuentra conformado por las 1400 matrices cuadradas presentes en la colección de matrices dispersas [8].

De las matrices, se midieron las características evaluadas en el trabajo [26], presentadas en la tabla 4.2. A su vez, se tomaron los tiempos de ejecución de cinco algoritmos de SpTrSv en el entorno experimental descrito en 4.3.

Estos algoritmos, descritos en 4.4, fueron elegidos debido a que representan el estado del arte, todos siendo el mejor algoritmo para algunas matrices. Estos se pueden dividir en dos categorías, aquellos con análisis y aquellos sin análisis.

Debido a la naturaleza secuencial de la SpTrSv, existen algoritmos que para poder aprovechar al máximo el paralelismo, requieren de una fase previa de análisis de la matriz a trabajar, para planificar la distribución de filas a procesar en paralelo. Este análisis suele tomar un tiempo considerable, pero puede ser beneficioso a largo plazo cuando se realizan varias iteraciones de SpTrSv sobre la misma matriz, como es el caso de la resolución de sistemas de ecuaciones por medio de métodos iterativos (como PCG, GMRES, BiCGSTAB, entre otros).

Con el fin de representar estas situaciones se evalúan los modelos en tres posibles instancias del problema, la primera replicando una sola corrida del

Modelo	Librería	Abreviación
Random Forest	Sklearn	rf
Extra Trees	Sklearn	et
Gradient Boosting basado en histogramas	Sklearn	hgb
Gradient Boosting	Sklearn	gb
AdaBoost con arboles de decisión	Sklearn	ada
Bagging con arboles de decision	Sklearn	bag
K-nearest neighbours	Sklearn	knn
Arboles de decision	Sklearn	dt
Gaussian Naive Bayes	Sklearn	gnb
Perceptron multicapa	Sklearn	mlp
Clasificador Passive Aggressive	Sklearn	pac
SVM con descenso estocástico de gradiente	Sklearn	sgd
Regresion Ridge	Sklearn	ridge
XGBoost	XGBoost	xgb
XGBoost con Random Forest	XGBoost	xgbrf
LightGBM	LightGBM	lgbm
CatBoost	CatBoost	cat

Tabla 4.1: Modelos comparados

Característica	Definición
n	Tamaño de la matriz
nnz	Cantidad de celdas distintas a cero
max nnz	Cantidad máxima de nnz en una fila
nnz promedio por fila	Promedio de nnz en todas las filas
devs standard de nnz	Desviación standard de los nnz en las filas
bandwidth promedio	Distancia promedio de la celda distinta a cero más lejana a la diagonal
n levels	Profundidad del grafo de dependencias de las filas
locality simple	Medición de paralelismo para el algoritmo simple
locality multi	Medición de paralelismo para el algoritmo multirow

Tabla 4.2: Características base evaluadas de las matrices

CPU	AMD Ryzen 5 5500 (6 cores)
GPU	NVIDIA GeForce RTX 3050 8GB
RAM	32 GB
SO	Ubuntu 18 con CUDA 10

Tabla 4.3: Entorno experimental utilizado para medir los tiempos de ejecución

Algoritmo	Requiere análisis?	Origen
Multirow	Sí	[3]
Yuenyeung (YY)	Sí	[36]
NaN	No	[3]
Simple	No	[37]
Order	No	[3]

Tabla 4.4: Algoritmos de SpTrSv evaluados

	1 corrida					
	Entrenamiento		Validación		Evaluación	
Algoritmo	Cantidad	Porcentaje	Cantidad	Porcentaje	Cantidad	Porcentaje
multi	0	-	0	-	0	-
yy	223	26.6 %	75	26.8 %	75	26.8 %
nan	473	56.3 %	157	56.1 %	157	56.1 %
simple	8	1.0 %	2	0.7 %	3	1.1 %
order	136	16.2 %	46	16.4 %	45	16.1 %
Total	840	100.0 %	280	100.0 %	280	100.0 %

Tabla 4.5: Distribución de mejor algoritmo en los conjuntos en 1 corrida

problema sobre la matriz, la siguiente con un análisis y diez corridas sobre la matriz, y por último un análisis y cien corridas.

Se entrenaron todos los modelos en 60 tests de Optuna que elige los metapárametros de cada modelo, entrenados usando las 9 características que se describen en 4.2.

Siguiendo buenas prácticas de aprendizaje automático [29], se dividen las mediciones en tres conjuntos, el primero, correspondiente al 60 % del total, se utiliza para entrenar, un 20 % de validación, para evaluar el modelo durante los tests de Optuna, y por último, un 20 % para la evaluación final, evitando sobreajustar al conjunto de validación. Resulta necesario que los conjuntos de validación y de evaluación reflejen la realidad, por lo que es importante que estos cuenten con una distribución de los mejores algoritmos similar al total.

Se elige una partición obtenida al azar para utilizar durante todas las pruebas, la cual sigue la distribución presentada en 4.5.

Es interesante destacar como la distribución cumple con ciertas condiciones razonables, por ejemplo, como que el porcentaje de matrices donde multi resulta el mejor se incrementa en gran medida con la cantidad de corridas dado que se compensa el tiempo añadido por el análisis.

	10 corridas					
	Entrenamiento		Validación		Evaluación	
Algoritmo	Cantidad	Porcentaje	Cantidad	Porcentaje	Cantidad	Porcentaje
multi	255	30.4 %	85	30.4 %	85	30.4 %
yy	181	21.6 %	61	21.8 %	60	21.4 %
nan	329	39.2 %	109	38.9 %	109	38.9 %
simple	8	1.0 %	2	0.7 %	3	1.1 %
order	67	8.0 %	23	8.2 %	23	8.2 %
Total	840	100.0 %	280	100.0 %	280	100.0 %

Tabla 4.6: Distribución de mejor algoritmo en los conjuntos en 10 corridas

	100 corridas					
	Entrenamiento		Validación		Evaluación	
Algoritmo	Cantidad	Porcentaje	Cantidad	Porcentaje	Cantidad	Porcentaje
multi	629	74.9 %	209	74.6 %	209	74.6 %
yy	49	5.8 %	17	6.1 %	16	5.7 %
nan	151	18.0 %	50	17.9 %	51	18.2 %
simple	2	0.2 %	1	0.4 %	1	0.4 %
order	9	1.1 %	3	1.1 %	3	1.1 %
Total	840	100.0 %	280	100.0 %	280	100.0 %

Tabla 4.7: Distribución de mejor algoritmo en los conjuntos en 100 corridas

Para evaluar si los modelos presentan resultados útiles, se utiliza como base para comparar el TAR esperado ($TARE$) de elegir al azar entre los algoritmos y el TAR de siempre elegir cada algoritmo, ambos evaluados en los conjuntos de validación y de evaluación, en cada cantidad de corridas.

Se puede ver en 4.8 como en 1 y 100 corridas, hay algoritmos que predominan y elegirlos siempre resulta bastante mejor que elegir aleatoriamente, mientras que en 10 corridas los TAR son mucho más balanceados. Estos TAR esperados son los objetivos a vencer con los modelos de aprendizaje automático.

Optuna permite optimizar más de un objetivo a la vez, por lo que se definieron dos objetivos para el entrenamiento. Por un lado, Optuna intenta optimizar el TAR para obtener el valor más bajo, mientras que por otro lado se define

	1 corrida	
	Validación	Evaluación
TARE	7.2078	8.7223
multi	30.1980	38.1876
yy	1.4180	1.5728
nan	0.8846	0.7357
simple	2.5541	2.2544
order	0.9842	0.8611

Tabla 4.8: TAR esperado y TAR de siempre elegir cada algoritmo en 1 corrida

	10 corridas	
	Validación	Evaluación
TARE	2.0580	2.2695
multi	2.7389	3.9257
yy	1.7158	1.7189
nan	1.2583	1.2215
simple	3.4579	3.2863
order	1.1194	1.1949

Tabla 4.9: TAR esperado y TAR de siempre elegir cada algoritmo en 10 corridas

	100 corridas	
	Validación	Evaluación
TARE	2.2303	2.5788
multi	0.2700	0.3347
yy	2.9571	2.5208
nan	1.7484	2.3250
simple	4.4491	5.8658
order	1.7268	1.8476

Tabla 4.10: TAR esperado y TAR de siempre elegir cada algoritmo en 100 corridas

como un objetivo "multi"donde intenta maximizar la precisión y minimizar el TAR simultáneamente.

Se señala con color en las tablas 4.11, 4.12 y 4.13 el modelo con el valor de TAR más bajo para cada posible realidad. Es interesante destacar cómo los modelos basados en gradient boosting (gb, hgb y xgb) suelen tener los mejores resultados, solo no siendo el mejor modelo en la tabla 4.13, pero por 0.0022, una diferencia mínima.

Modelo	TAR	Precisión	Objetivo
ada	0.0599	0.8500	multi
bag	0.0572	0.8679	multi
cat	0.0528	0.8607	tar
dt	0.1325	0.8000	multi
et	0.0819	0.8429	tar
gb	0.0396	0.8786	multi
gnb	0.6159	0.3857	multi
hgb	0.0489	0.8714	multi
knn	0.1778	0.8179	multi
lgbm	0.0505	0.8714	multi
mlp	0.0701	0.8429	multi
pac	0.1217	0.7821	multi
rf	0.0534	0.8643	multi
ridge	0.5944	0.6321	multi
sgd	0.1566	0.7643	tar
xgb	0.0421	0.8750	multi
xgbrf	0.0539	0.8679	multi

Tabla 4.11: Un análisis y una corrida

Modelo	TAR	Precisión	Objetivo
ada	0.1342	0.7250	multi
bag	0.0855	0.7643	tar
cat	0.0916	0.7250	tar
dt	0.2082	0.6571	multi
et	0.0864	0.7500	multi
gb	0.1042	0.7179	tar
gnb	0.7472	0.4036	multi
hgb	0.0880	0.7536	multi
knn	0.3099	0.6107	multi
lgbm	0.1620	0.7464	tar
mlp	0.4288	0.6607	multi
pac	0.3764	0.6036	multi
rf	0.1455	0.7679	multi
ridge	0.9747	0.4786	multi
sgd	0.3787	0.5929	multi
xgb	0.0766	0.7571	tar
xgbrf	0.1093	0.7036	tar

Tabla 4.12: Un análisis y diez corridas

Modelo	TAR	Precisión	Objetivo
ada	0.2164	0.8286	tar
bag	0.2087	0.8071	tar
cat	0.2177	0.8179	tar
dt	0.2089	0.7750	multi
et	0.1855	0.8286	multi
gb	0.1877	0.8214	tar
gnb	0.9401	0.3679	multi
hgb	0.2235	0.8179	multi
knn	0.3183	0.7750	multi
lgbm	0.1880	0.8179	tar
mlp	0.2611	0.7607	tar
pac	0.3388	0.7500	multi
rf	0.1866	0.8214	tar
ridge	0.3359	0.7429	multi
sgd	0.2653	0.6143	tar
xgb	0.2236	0.8286	tar
xgbrf	0.2290	0.8179	tar

Tabla 4.13: Un análisis y cien corridas

A su vez, para la mayoría de modelos, configurar Optuna para que intente optimizar los dos valores a la vez (precisión y TAR) resulta lo mejor, seguramente debido a que es muy difícil tener un TAR muy bueno pero una precisión muy baja, por lo que al estar ambos tan entrelazados, el optimizarlos en conjunto puede llevar al modelo a corregir sus principales problemas, si es que esta errando mucho o agregando mucho tiempo.

Esto no es el caso para los algoritmos de gradient boosting, donde en la mayoría de situaciones, optimizar solo el TAR los hizo obtener los mejores valores.

4.1.1. Resultados combinando algoritmos

Con el fin de obtener mejor TAR, se utiliza el preprocesamiento que combina las etiquetas de aquellas matrices donde los mejores algoritmos tienen tiempos de ejecución muy similares. Para esto se le agrega a Optuna la opción de elegir si combinar algoritmos (o no) a ciertos márgenes definidos: 0.05, 0.10 y 0.25.

Estos márgenes se interpretan como el máximo valor donde se combinarán los dos mejores algoritmos de una matriz, y resulta de evaluar la siguiente expresión:

$$\frac{SegundoMejorTiempo - MejorTiempo}{MejorTiempo} \tag{4.1}$$

Es importante destacar que el entrenamiento con Optuna es inherentemente aleatorio, por lo que ocurre que el mismo modelo, aún sin combinar los algoritmos, resulte en peor o mejores valores que cuando no era opción combinar.

Modelo	TAR	Precisión	Objetivo	Margen de combinación
ada	0.0702	0.8929	tar	0.25
bag	0.0572	0.8643	multi	-
cat	0.0559	0.8536	multi	-
dt	0.1213	0.8357	multi	0.25
et	0.0800	0.8571	multi	0.05
gb	0.0512	0.9214	multi	0.10
gnb	0.3559	0.5821	tar	0.10
hgb	0.0541	0.8857	tar	0.05
knn	0.1778	0.8179	multi	-
lgbm	0.0528	0.8679	multi	-
mlp	0.1485	0.8714	multi	0.10
pac	0.2356	0.7821	multi	0.25
rf	0.0618	0.8357	tar	-
ridge	0.5620	0.7357	multi	0.25
sgd	0.1498	0.8000	multi	0.10
xgb	0.0566	0.8571	tar	-
xgbrf	0.0536	0.8714	tar	-

Tabla 4.14: Una corrida con combinación de algoritmos al entrenar

Para calcular la precisión de un modelo con combinación, si este predice que un algoritmo era el mejor y ese o la combinación de ese algoritmo con otro son los mejores, la predicción se considera correcta. También se considera correcto si predice una combinación de algoritmos y uno de los dos algoritmos es el de mejor tiempo.

Las etiquetas combinadas siguen la forma «$algoritmoA - algoritmoB$», ordenados alfabéticamente y es el primero del par el que se utiliza para calcular el TAR.

Esto es un gran punto de mejora y es posible que los TAR y la precisión de estos modelos no reflejen los mejores resultados posibles debido a que se podría optimizar por medio de heurísticas las elecciones de cuál algoritmo utilizar si el modelo predice que lo mejor es una combinación de algoritmos. Por ejemplo, se podría definir una simple heurística que en vez de ordenar los algoritmos alfabéticamente, se los ordene por la cantidad de veces en que resulta el mejor (como en la tabla 4.5) o según el TAR esperado de siempre elegirlos. Esto debería afectar positivamente los TAR en general, dado que es una elección más consciente del contexto y que estadísticamente debería brindar mejores resultados.

Es importante destacar que la combinación de algoritmos, además, agrega un cierto tiempo de preprocesamiento de los datos para el entrenamiento, dado que complica el etiquetado inicial, pero al momento de hacer las predicciones, no tiene efecto.

En los resultados (las tablas 4.14, 4.15 y 4.16) se puede notar cómo no combinar (aquellas filas marcadas con un guion en los márgenes de combinación)

Modelo	TAR	Precisión	Objetivo	Margen de combinación
ada	0.1506	0.7393	multi	0.05
bag	0.1346	0.7250	tar	0.05
cat	0.1485	0.8429	multi	0.05
dt	0.1428	0.7214	tar	0.05
et	0.1625	0.8321	tar	0.05
gb	0.1013	0.7321	multi	-
gnb	0.8390	0.6500	multi	0.10
hgb	0.0865	0.7321	tar	-
knn	0.2827	0.6143	tar	-
lgbm	0.1703	0.8500	multi	0.05
mlp	0.0908	0.7000	tar	-
pac	0.3868	0.5393	tar	-
rf	0.1414	0.8250	tar	0.05
ridge	1.2775	0.5714	multi	0.10
sgd	0.2589	0.6500	multi	0.05
xgb	0.1099	0.7214	tar	-
xgbrf	0.1447	0.8214	multi	0.05

Tabla 4.15: Diez corridas con combinación de algoritmos al entrenar

Modelo	TAR	Precisión	Objetivo	Margen de combinación
ada	0.2125	0.8964	multi	0.05
bag	0.1836	0.9000	multi	0.05
cat	0.1906	0.9107	multi	0.25
dt	0.2059	0.7893	tar	-
et	0.1920	0.8321	multi	-
gb	0.1909	0.8214	multi	-
gnb	0.3665	0.8500	multi	0.05
hgb	0.2273	0.8179	multi	-
knn	0.2485	0.8500	tar	0.10
lgbm	0.2154	0.8286	tar	-
mlp	0.2035	0.9000	multi	0.25
pac	0.2495	0.8607	multi	0.25
rf	0.1867	0.8179	tar	-
ridge	0.3360	0.8071	multi	0.05
sgd	0.2524	0.8571	tar	0.10
xgb	0.1884	0.9107	tar	0.25
xgbrf	0.2280	0.8214	tar	-

Tabla 4.16: Cien corridas con combinación de algoritmos al entrenar

sigue siendo una opción fuerte, pero para varios modelos, combinar les permite obtener mejores resultados. Principalmente en el escenario de 10 corridas, sucede que combinar con un margen estricto es lo mejor, mientras que en las demás situaciones, un margen más liviano, como es 0.25, funciona. Es muy posible que en estos casos existan diferencias tan grandes entre los tiempos de ejecución, por lo que si los márgenes no son altos, se observen pocas combinaciones y los modelos no se aprovechen de esto, y estos valores serían los mínimos razonables, donde algoritmos muy distantes no se junten, pero sí los verdaderamente similares.

4.2. Características

Para evaluar las características utilizadas, se definen cuatro conjuntos distintos y Optuna se encarga de elegir cuál utilizar.

Estos conjuntos, presentados en la sección 3.2.2 se definieron considerando los tiempos que toman en obtenerse, con el fin de evaluar si con un conjunto limitado de características, igual se obtienen buenos resultados.

Lo más destacable de estos resultados, presentados en las tablas 4.18, 4.19 y 4.20, son el hecho de que para el escenario de 100 corridas, el TAR obtenido es el mejor hasta el momento, usando todas las mejoras planteadas, con el objetivo de minimizar solo el TAR, combinando los algoritmos y además usando el conjunto de características completo, que incluye las nuevas características.

Conjunto	Abreviación
Original	Original
Rápido Original	O(c)
Completo	Nuevo
Rápido Extendido	Rápido

Tabla 4.17: Abreviaturas utilizadas para representar cada conjunto

Modelo	TAR	Precisión	Objetivo	Margen de combinación	Características
ada	0.0744	0.8321	multi	-	nuevo
bag	0.0552	0.8607	tar	-	original
cat	0.0737	0.8964	multi	0.10	nuevo
dt	0.0917	0.8393	multi	0.05	original
et	0.0634	0.8964	multi	0.10	original
gb	0.0558	0.9143	multi	0.10	original
gnb	0.3568	0.5821	multi	0.10	original
hgb	0.0646	0.8786	tar	0.05	original
knn	0.1712	0.8679	multi	0.25	o(c)
lgbm	0.0525	0.8821	tar	0.05	nuevo
mlp	0.1173	0.8143	tar	-	original
pac	0.2356	0.7821	multi	0.25	original
rf	0.0597	0.8429	multi	-	original
ridge	0.5425	0.6893	multi	0.05	nuevo
sgd	0.1894	0.7964	tar	0.05	original
xgb	0.0661	0.8786	multi	0.05	nuevo
xgbrf	0.0573	0.8607	multi	-	original

Tabla 4.18: Una corrida con combinación y distintos conjuntos de características

Modelo	TAR	Precisión	Objetivo	Margen de combinación	Características
ada	0.1621	0.8214	multi	0.05	nuevo
bag	0.1174	0.8036	multi	0.05	nuevo
cat	0.1541	0.7357	tar	-	original
dt	0.1496	0.7429	tar	0.05	original
et	0.0861	0.7464	multi	-	original
gb	0.1316	0.8071	tar	0.05	nuevo
gnb	0.5295	0.5286	tar	0.10	o(c)
hgb	0.1681	0.8107	tar	0.05	nuevo
knn	0.2726	0.6607	multi	0.05	original
lgbm	0.1712	0.7357	tar	-	original
mlp	0.2242	0.6143	tar	-	o(c)
pac	0.2620	0.6071	multi	0.05	o(c)
rf	0.0823	0.8143	multi	0.05	nuevo
ridge	0.9650	0.5429	tar	0.05	original
sgd	0.2773	0.6357	multi	0.10	o(c)
xgb	0.1303	0.7321	tar	-	nuevo
xgbrf	0.0988	0.7107	tar	-	nuevo

Tabla 4.19: Diez corridas con combinación y distintos conjuntos de características

Modelo	TAR	Precisión	Objetivo	Margen de combinación	Características
ada	0.1763	0.8214	multi	-	nuevo
bag	0.1850	0.8929	multi	0.05	original
cat	0.2170	0.8964	tar	0.05	nuevo
dt	0.1500	0.8857	tar	0.05	nuevo
et	0.2021	0.9036	tar	0.25	rápido
gb	0.1809	0.9036	multi	0.25	nuevo
gnb	0.3665	0.8500	tar	0.05	original
hgb	0.2205	0.8214	tar	-	original
knn	0.2368	0.8571	multi	0.10	original
lgbm	0.1821	0.8929	tar	0.10	nuevo
mlp	0.2458	0.8536	tar	0.05	original
pac	0.2673	0.8536	tar	0.25	original
rf	0.1866	0.8214	multi	-	original
ridge	0.3360	0.8071	multi	0.05	original
sgd	0.2434	0.5857	multi	-	o(c)
xgb	0.1898	0.9071	multi	0.25	original
xgbrf	0.2309	0.8357	tar	-	nuevo

Tabla 4.20: Cien corridas con combinación y distintos conjuntos de características

A su vez, se evaluaron los 2 mejores modelos de cada escenario con 20 tests de Optuna y se graficaron los mejores TAR obtenidos en las gráficas 4.1, 4.2 y 4.3. En estos se puede ver como en el escenario de 1 corrida, los conjuntos más limitados dan resultados muy inferiores a los resultados de los conjuntos más completos. Por otro lado, en los escenarios con más corridas, los modelos tienden a dar resultados similares para todos los conjuntos.

Un detalle importante de los conjuntos limitados es que, al consistir en las propiedades calculables en tiempos despreciables, no cuentan con la característica *n_levels*, que es la que más explícitamente determina el posible paralelismo obtenible y es muy posiblemente una de las principales propiedades para determinar los algoritmos.

De todas formas, resulta interesante que el conjunto rápido presente resultados tan similares al resto de conjunto, porque teóricamente las mediciones necesarias para el conjunto rápido deberían tomar mucho menos tiempo que para los conjuntos completos.

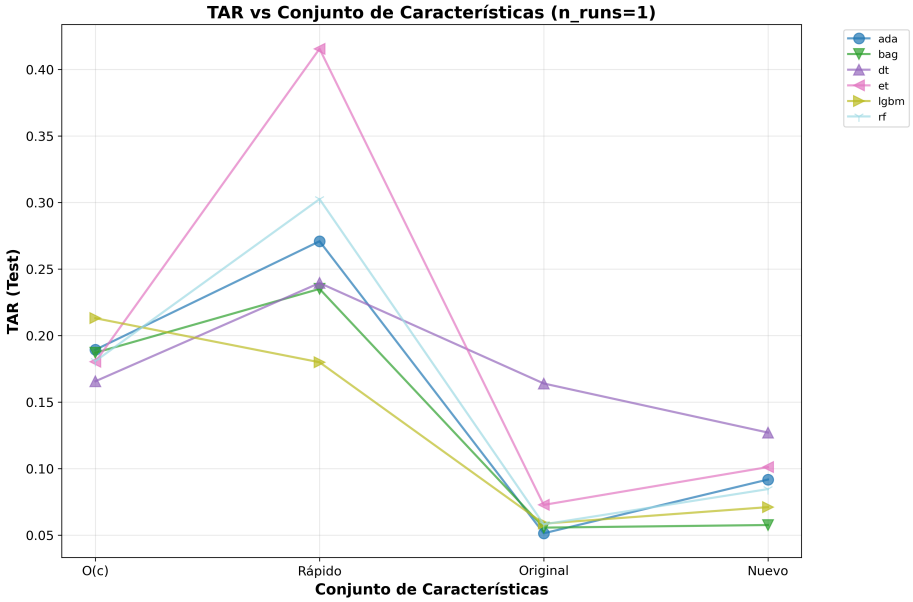


Figura 4.1: Mejor TAR de los mejores 6 modelos con cada conjunto, para 1 corrida

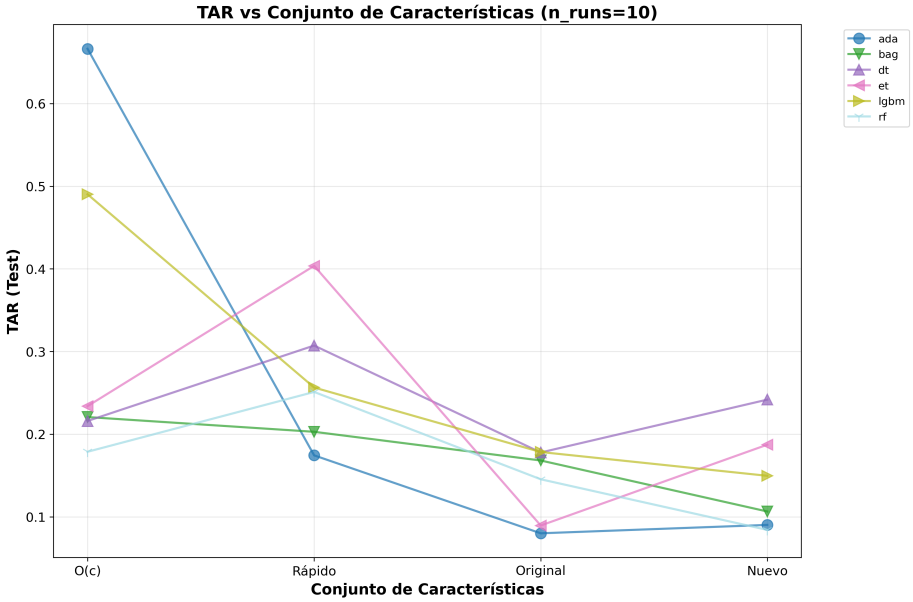


Figura 4.2: Mejor TAR de los mejores 6 modelos con cada conjunto, para 10 corridas

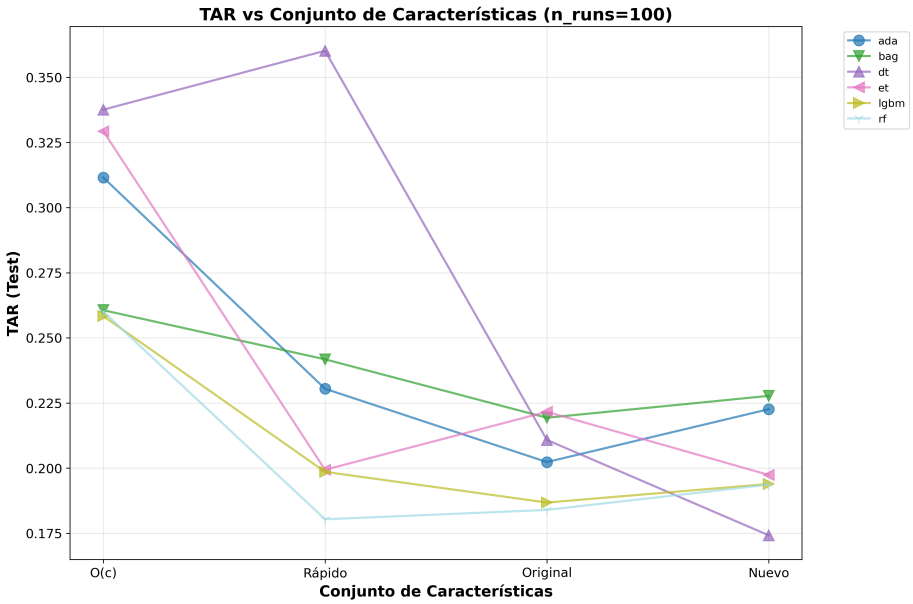


Figura 4.3: Mejor TAR de los mejores 6 modelos con cada conjunto, para 10 corridas

4.3. Generación artificial de matrices

Lo primero a evaluar al momento de utilizar las matrices para entrenar, es que cumplan con ciertas condiciones para que repliquen el comportamiento de las reales. El proceso de generación de matrices se basa en dos procesos que deben funcionar correctamente: el proceso de downscale-upscale y el proceso de generación de imágenes.

4.3.1. Proceso de downscale-upscale

Para evaluar el proceso de downscale-upscale, se toman 200 matrices reales y se les aplica el downscale a 128x128 y luego se les aplica el upscale de nuevo a su tamaño original. Si el proceso es correcto, se debe cumplir que la cantidad de elementos no nulos se mantiene cercana y los patrones espaciales también se mantienen.

A continuación se eligen al azar 3 matrices que muestran las limitaciones de la implementación. Al utilizar enteros entre 0 y 255 para representar las densidades, ocurre que regiones con muy pocos elementos distintos a cero se redondea a 0 y se pierden regiones. Esto se ve de forma más clara en las matrices más grandes, como la de la figura 4.4 en el cuadrante inferior derecho.

Para solventar este problema, se evalúa a su vez con diferentes rangos, y por ejemplo, al utilizar valores de 0 a 1023. Esto arroja mejores resultados en casos

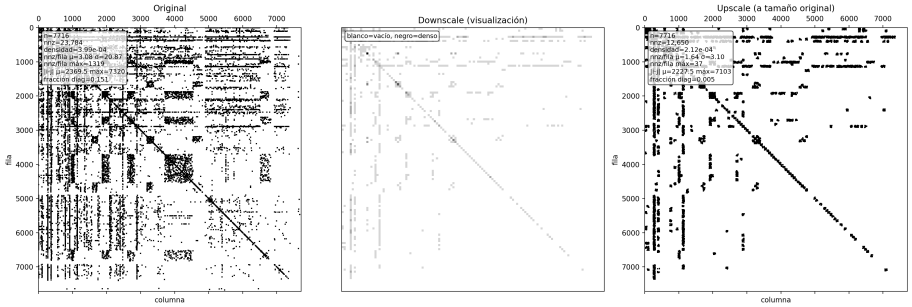


Figura 4.4: Matriz *as-735* con la matriz resultado del downscale con valores entre 0 a 255

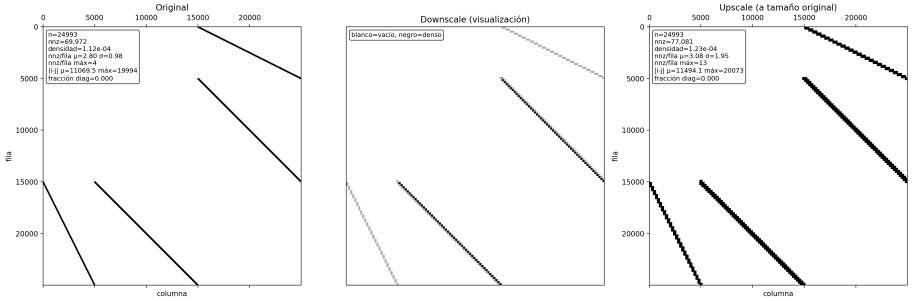


Figura 4.5: Matriz *dtoc* con la matriz resultado del downscale con valores entre 0 a 255

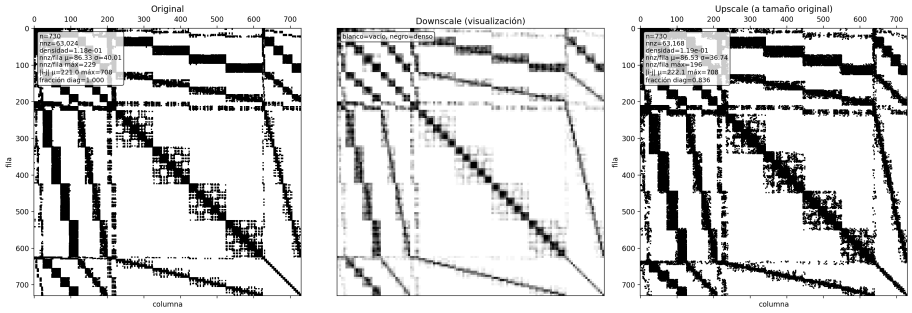


Figura 4.6: Matriz *dendrimer* con la matriz resultado del downscale con valores entre 0 a 255

Matriz	Elementos distintos a cero
Original	23784
Upscalada con 0-255	12650
Upscalada con 0-1023	23674

Tabla 4.21: Cantidad de elementos distintos a cero al aplicar el proceso de downscale-upscale a la matriz *AS-735*

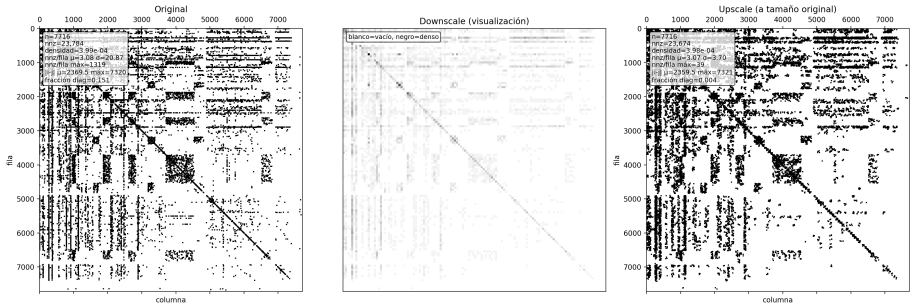


Figura 4.7: Matriz *as-735* con la matriz resultado del downscale con valores entre 0 a 1023

como 4.7 donde se puede ver cómo al utilizar un rango mayor (cuatro veces más descriptivo en este caso) se obtiene una matriz más similar a la original, no solo visualmente sino que a la vez la cantidad de elementos distintos a cero terminó siendo casi igual, como se ve en la figura 4.21.

Utilizar un rango mayor permite mayor expresividad, pero esta fue una mejora que recién se pudo implementar después de generar el conjunto de entrenamiento y debido al tiempo que toma generar el conjunto, se decidió seguir utilizando el rango de 0 a 255.

Otro parámetro que se puede variar, es el tamaño «destino» al que se realiza

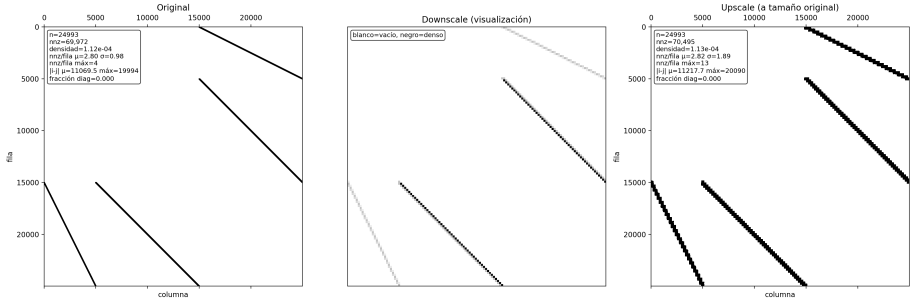


Figura 4.8: Matriz *dtoc* con la matriz resultado del downscale con valores entre 0 a 1023

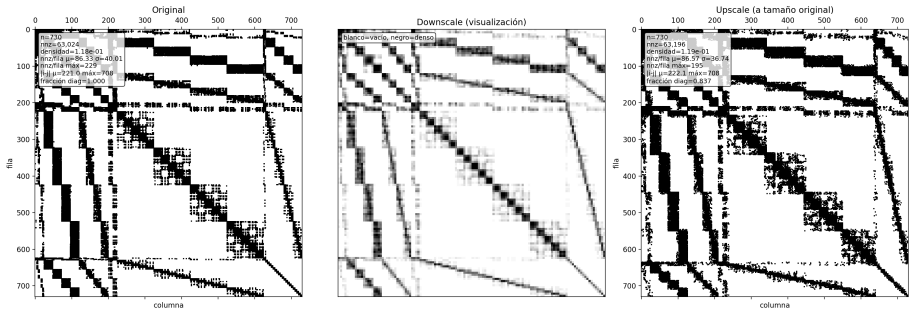


Figura 4.9: Matriz *dendrimer* con la matriz resultado del downscale con valores entre 0 a 1023

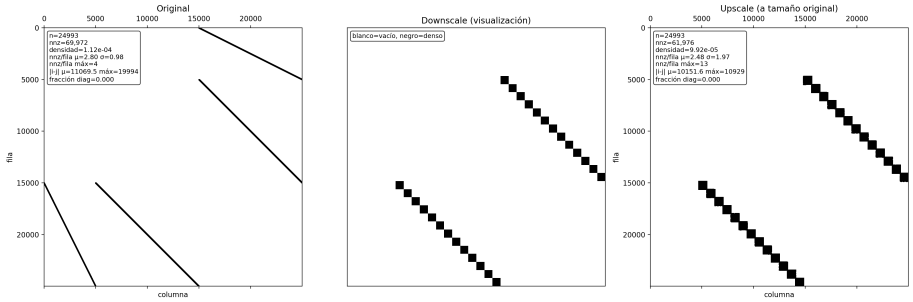


Figura 4.10: Matriz *dtoc* resultante del downscale a una matriz 32×32 con pixeles en el rango 0-255

el downscale. Este impacta directamente en la velocidad de entrenamiento de la GAN, por lo que utilizar tamaños grandes resulta inconveniente. Tamaños pequeños resultan en matrices poco expresivas, perdiendo mucha información estructural al quedar muchas regiones englobadas en la misma región de la matriz del downscale. A su vez, al realizar el upscale de matrices pequeñas se obtienen matrices muy cuadriculadas, ambos problemas evidentes en la figura 4.10.

Se buscó un balance entre expresividad y velocidad, por lo que para la generación se fija el tamaño de las imágenes en *128x128*.

4.3.2. Entrenamiento de la GAN

Durante el entrenamiento de la GAN, se controla el progreso de la misma inspeccionando visualmente matrices generadas por el generador de forma periódica. Este debería descubrir patrones repetidos en las matrices reales, como la diagonal principal o diagonales secundarias. Para el conjunto de entrenamiento primero se realiza el downscale a *128x128* con rango 0-255 de las 1400 matrices cuadradas provenientes de SuiteSparse. En este conjunto se detectó que

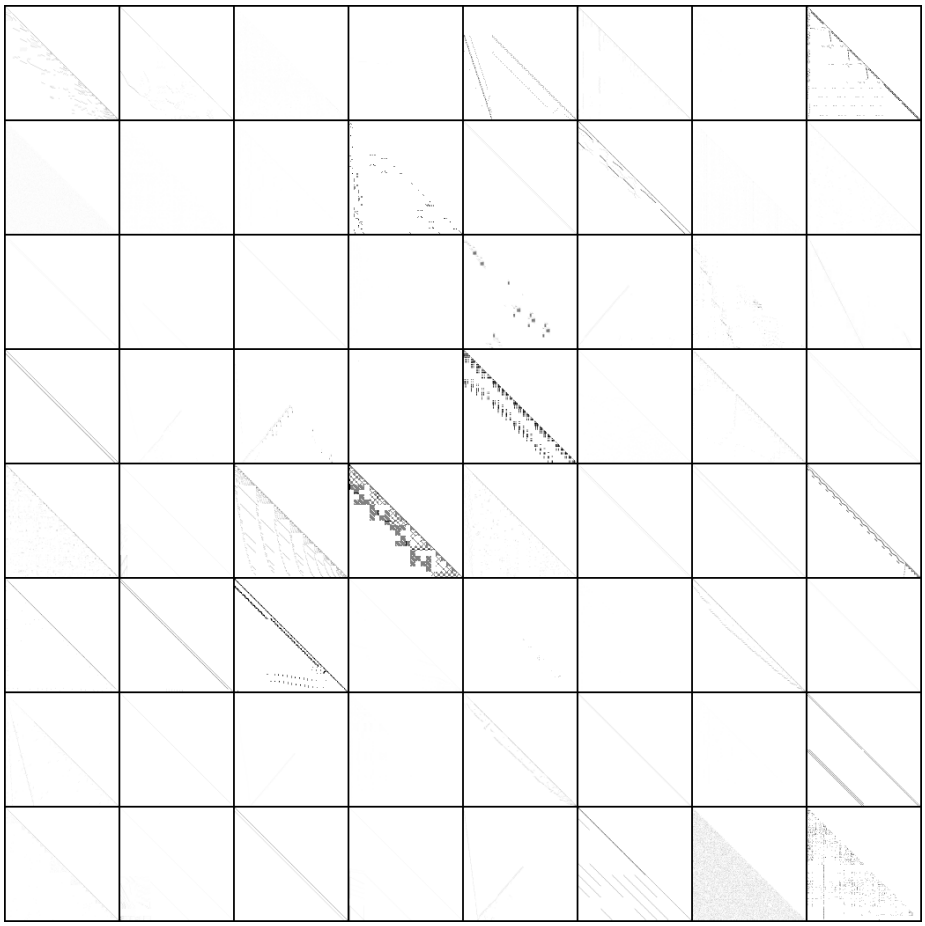


Figura 4.11: Subconjunto de 64 imágenes de matrices utilizadas para entrenar la GAN

muchas se veían exactamente iguales, por lo que la GAN estaba aprendiendo de imágenes repetidas. Esto se evitó con un preprocesamiento del conjunto de imágenes que calcula la similitud entre las imágenes y se queda con aquellas lo suficientemente distintas. Al finalizar este proceso se obtuvieron alrededor de 600 imágenes.

En la figura 4.11 se pueden ver 64 matrices elegidas al azar del conjunto de entrenamiento. Muchas visualmente parecen vacías, pero en realidad cuentan con algunos valores distintos a cero muy bajos que resultan en grises no diferenciables a simple vista.

La GAN es entrenada durante 150 generaciones. Se puede ver en 4.12 cómo para la generación número 19, este ya detectó las diagonales, pero aún genera fondos demasiado densos. Por otro lado, en la figura 4.13 se puede ver como

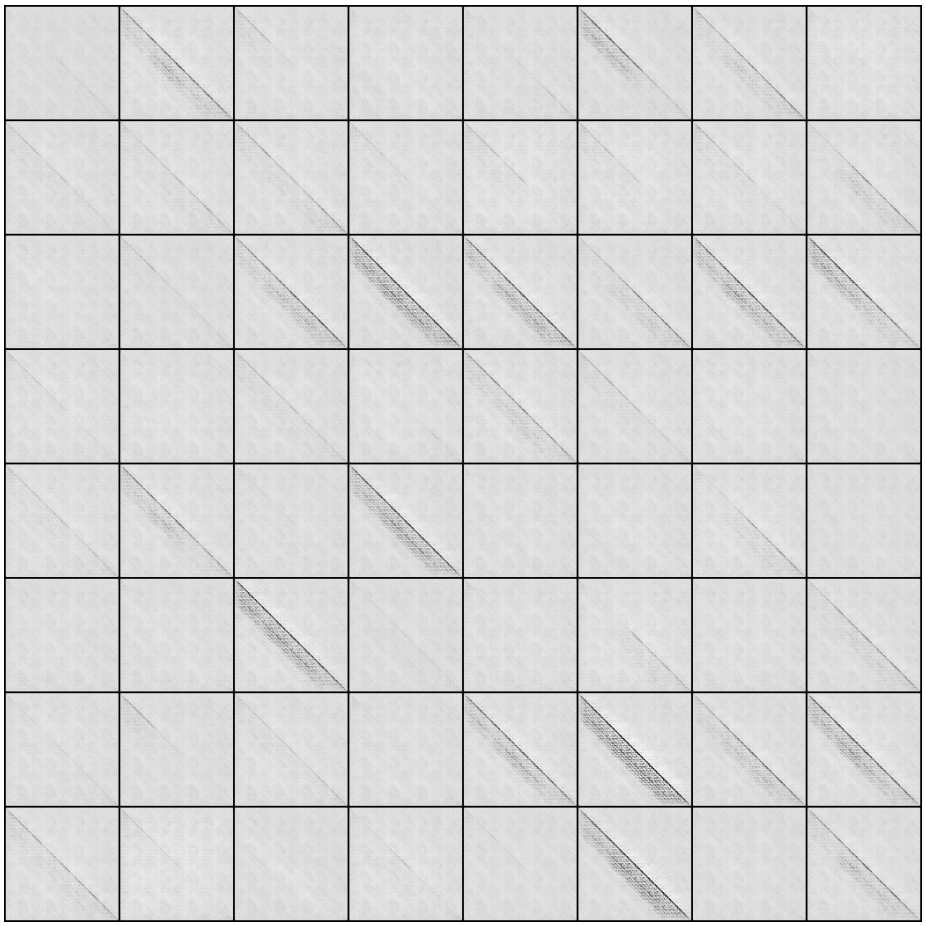


Figura 4.12: Matrices generadas por el Generador en la generación 19

las matrices ahora ya tienen fondos completamente vacíos e intenta generar patrones varios en el triángulo inferior de las matrices. Esta generación es la que se consideró como mejor y es la que fue utilizada para generar las matrices finales, dado que es la que minimizó las funciones de pérdida de ambas redes, es decir, es la generación donde el Generador realizó el mejor trabajo engañando al Discriminador.

4.3.3. Evaluación de los valores estadísticos de las matrices sintéticas

Una vez que se tiene entrenado el generador, este se utiliza para generar la cantidad de matrices que se desee. El proceso de upscale y los posibles tamaños deseados resultan el principal cuello de botella del proceso, dado que este se

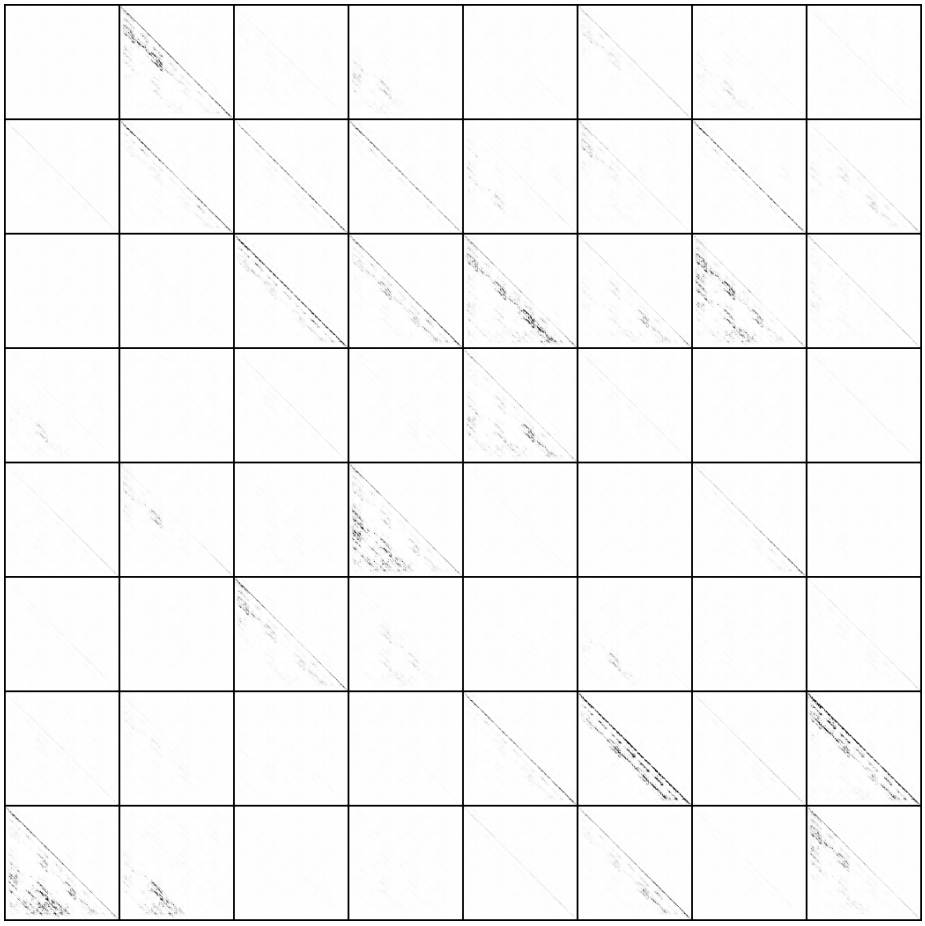


Figura 4.13: Matrices generadas por el Generador en la generación 87

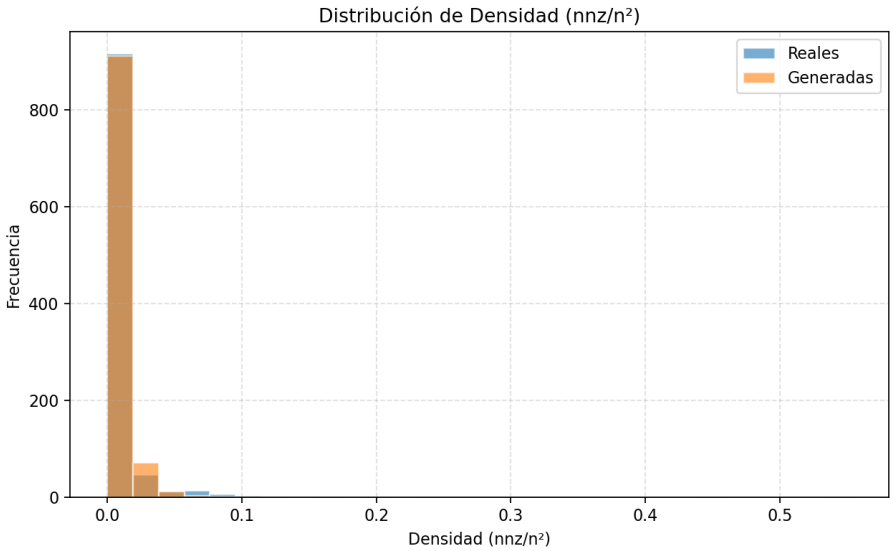


Figura 4.14: Histograma de las densidades

ve limitado por las capacidades de cómputo y la memoria disponible. Generar una matriz de más de 100,000 × 100,000 frecuentemente resulta en una matriz demasiado grande con demasiados elementos distintos de cero, por lo que no se pudieron procesar.

Para verificar que las matrices sintéticas son útiles para el entrenamiento, no solo basta con la inspección visual, sino que lo deseado es que si se miden las mismas métricas en las matrices reales y en las sintéticas, estas sigan la misma distribución.

Primero, al ser fácilmente controlable y al no desear sesgar las matrices para ciertos algoritmos, se implementa un generador de números aleatorios que replica la distribución de los tamaños de las matrices y este es el que decide los tamaños a los cuales realizar el upscale.

Se tomaron aleatoriamente 1000 matrices reales y se generaron 1000 matrices. A estas se les miden distintas métricas y se grafican histogramas de frecuencia para evaluar las distribuciones.

Se utilizan métricas similares a las utilizadas en los conjuntos de características de la sección 3.2.2.

Por un lado, en el gráfico 4.14 las distribuciones de densidad coinciden plenamente, por lo que las matrices generadas no resultan con más elementos distintos a cero de los esperados.

Por otro lado, se puede ver en 4.15 cómo las matrices generadas suelen tener la diagonal principal mucho más vacía que las matrices reales. Esto es esperable, dado que el posicionamiento de los elementos dentro de cada región es totalmente aleatorio, por lo que las probabilidades de que terminen elementos en la diagonal

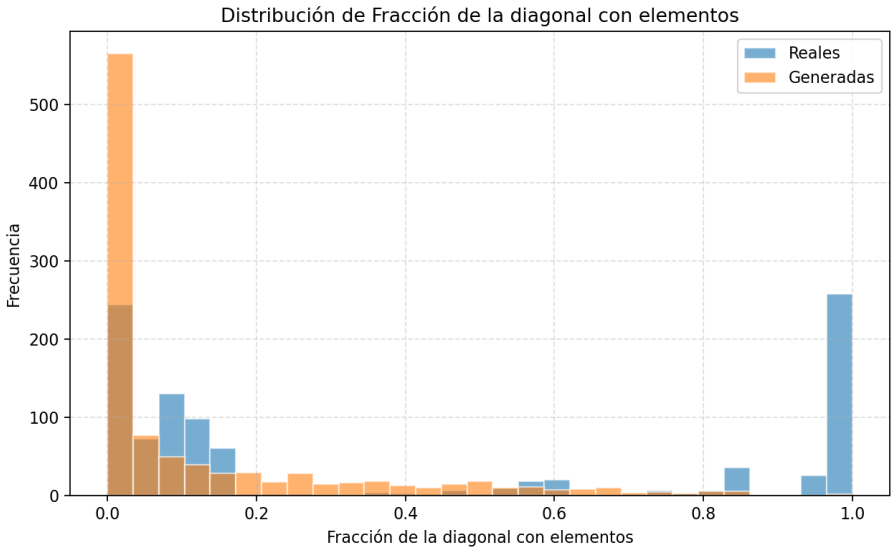


Figura 4.15: Histograma de la fracción de la diagonal principal con elementos distintos a cero

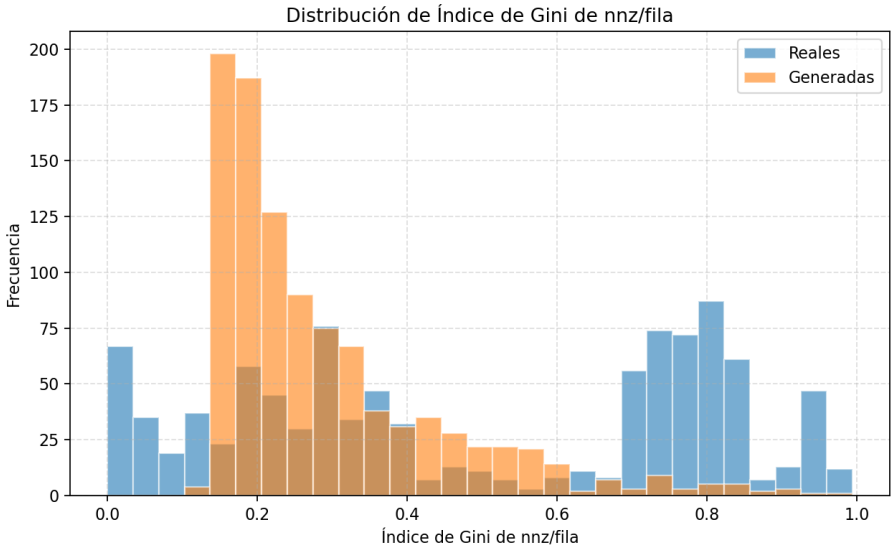


Figura 4.16: Histograma de los valores de Gini de las cantidades de elementos distintos a cero en cada fila

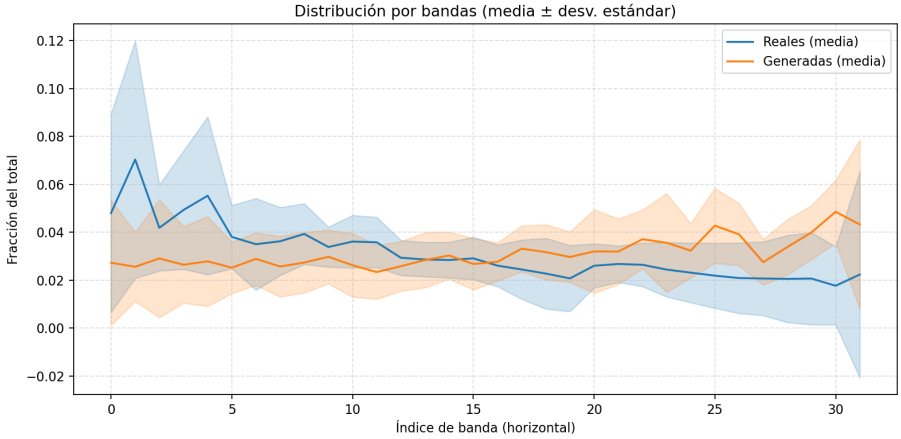


Figura 4.17: Porcentajes promedio de las distribuciones de elementos en 32 bandas

	Reales		Sintéticas	
Algoritmo	Cantidad	Porcentaje	Cantidad	Porcentaje
multi	0	-	0	-
yy	373	26.62 %	0	-
nan	786	56.10 %	3328	95.09 %
simple	15	1.07 %	94	2.69 %
order	227	16.20 %	78	2.23 %

Tabla 4.22: Distribución de los mejores algoritmos en las matrices generadas, en 1 corrida

son bajas, mientras que en la realidad la diagonal tiene una probabilidad más alta de contar con elementos.

En el gráfico de la distribución de los índices de Gini (figura 4.16), se puede ver cómo claramente siguen una distribución distinta. Las matrices generadas tienden a índices de Gini más bajos, lo cual se interpreta como que las filas tienden a ser más parejas que en las matrices reales, siguiendo una distribución aparentemente uniforme.

Esto se acompaña del gráfico 4.17, el cual indica que en las matrices reales, la fracción del total de elementos que cae en cada banda tiende a decaer a través de las bandas, mientras que en las generadas se mantienen más estable.

Por último, en las tablas 4.22, 4.23 y 4.24 se toman las 3500 matrices sintéticas utilizadas para entrenar y se comparan los resultados de los tiempos de los algoritmos, y se evalúa la distribución de los mismos.

De las distribuciones de los algoritmos se destaca que sigue un patrón que también ocurre con las matrices reales, donde con un bajo número de corridas, nan es el mejor algoritmo y conforme aumentan las corridas, multi comienza a

	Reales		Sintéticas	
Algoritmo	Cantidad	Porcentaje	Cantidad	Porcentaje
multi	425	30.34 %	4	0.11 %
yy	302	21.56 %	0	-
nan	546	38.97 %	3324	94.97 %
simple	15	1.07 %	94	2.69 %
order	113	8.07 %	78	2.23 %

Tabla 4.23: Distribución de los mejores algoritmos en las matrices generadas, en 10 corridas

	Reales		Sintéticas	
Algoritmo	Cantidad	Porcentaje	Cantidad	Porcentaje
multi	1047	74.73 %	1287	36.77 %
yy	82	5.85 %	0	-
nan	252	17.99 %	2114	60.40 %
simple	5	0.36 %	62	1.77 %
order	15	1.07 %	37	1.06 %

Tabla 4.24: Distribución de los mejores algoritmos en las matrices generadas, en 100 corridas

ganar.

Es cierto que multi en 100 corridas no alcanza a ser el mejor algoritmo, como si ocurre en las matrices reales. Esto puede deberse a una gran cantidad de factores, pero es posible que se deba al hecho de que las matrices sintéticas tienen menos dependencias entre filas y las filas suelen ser menos densas, por lo que la planificación de cómo distribuir el trabajo resulta menos útil y los tiempos de ejecución de multi no resultan considerablemente mejores que los de nan

De todas formas, ningún resultado es muy desalentador, en primer lugar, porque ninguna métrica está totalmente alejada, y aun si fuera así, al ser la generación un procedimiento inherentemente aleatorio, con suficiente tiempo y poder de cómputo se podría generar un gran número de matrices y descartar todas las necesarias para que las distribuciones sean exactamente iguales. A su vez, en cierta medida es bueno que las matrices sintéticas no sean exactamente iguales que las reales, para agregar variedad al conjunto y poder detectar patrones que posiblemente no se presentan en el conjunto real disponible, pero sí podrían ocurrir.

4.3.4. Entrenamiento de los modelos con matrices artificiales

El objetivo de generar las matrices, es aumentar el conjunto de entrenamiento de los modelos para mejorar los resultados, principalmente para aquellos mo-

Modelo	TAR	Precisión	Objetivo	Margen de combinación	Características
ada	0.0621	0.9000	tar	0.10	original
bag	0.0659	0.8571	multi	-	original
dt	0.1394	0.7893	multi	-	original
et	0.0949	0.8179	multi	-	original
gb	0.0689	0.9000	tar	0.10	original
gnb	0.3590	0.6750	tar	0.10	original
hgb	0.0748	0.8964	tar	0.10	original
knn	0.2009	0.7286	tar	-	o(c)
mlp	0.1008	0.7964	tar	-	original
pac	0.2080	0.8214	multi	0.25	original
rf	0.0645	0.8607	multi	-	original
ridge	0.6643	0.6321	multi	-	nuevo
sgd	0.1971	0.7643	tar	0.05	original
xgb	0.0691	0.8714	tar	0.05	original
xgbrf	0.0580	0.8571	tar	-	original

Tabla 4.25: Una corrida con combinación, distintos conjuntos de características y con matrices sintéticas

delos que pueden requerir de una mayor cantidad de datos para poder detectar patrones.

Para no sesgar los resultados a las matrices sintéticas, el conjunto de entrenamiento es aumentado con los datos provenientes de solo 3500 matrices generadas. Los conjuntos de validación y de evaluación se mantienen exactamente iguales a los utilizados anteriormente para permitir comparar los resultados de forma directa.

Mientras que en el escenario de una corrida, presentado en la tabla 4.25, se ve cómo se obtuvieron resultados cercanos a los obtenidos sin matrices sintéticas, pero que no logran ser superiores, en los otros dos escenarios se obtuvieron resultados mejores que todos los anteriores (tablas 4.26 y 4.27).

Lo principal a destacar de esto es el hecho de que se pueden considerar realistas las matrices sintéticas. Es razonable pensar que si las matrices generadas no presentaran patrones similares a las matrices reales, los modelos, al ser evaluados solo en matrices reales, no serían capaces de dar resultados mejores que las líneas base, porque estarían basándose en información errónea.

Por otro lado, es interesante destacar cómo en el escenario de 10 corridas, el mejor modelo resulta el que se basa en un perceptrón multicapa, conocido como red neuronal, el cual suele requerir mucha información para converger a un modelo útil, algo que claramente no estaba obteniendo del conjunto de matrices reales.

Finalmente, en el escenario de 100 corridas, el hecho de que el mejor modelo sea el k-nearest neighbours utilizando el conjunto rápido resulta prometedor, dado que este es un modelo inherentemente rápido para hacer predicciones y, además, el conjunto rápido no requiere de las métricas complejas de calcular.

Modelo	TAR	Precisión	Objetivo	Margen de combinación	Características
ada	0.0966	0.8429	tar	0.05	original
bag	0.1442	0.7679	tar	0.05	nuevo
dt	0.2071	0.7321	multi	0.05	original
et	0.1175	0.7964	tar	0.05	original
gb	0.1837	0.8500	multi	0.05	original
gnb	0.5984	0.3821	multi	-	original
hgb	0.1953	0.8286	tar	0.05	original
knn	0.1972	0.6893	multi	0.05	original
mlp	0.0715	0.7429	multi	-	original
pac	0.4834	0.5500	tar	-	original
rf	0.1592	0.8357	multi	0.05	original
ridge	1.2445	0.4071	multi	-	original
sgd	0.2264	0.6393	tar	-	original
xgb	0.1484	0.7107	tar	-	nuevo
xgbrf	0.1435	0.7821	tar	0.05	original

Tabla 4.26: Diez corridas con combinación, distintos conjuntos de características y con matrices sintéticas

Modelo	TAR	Precisión	Objetivo	Margen de combinación	Características
ada	0.2052	0.8071	tar	-	original
bag	0.1920	0.9179	multi	0.10	nuevo
dt	0.1811	0.9000	multi	0.25	original
et	0.1864	0.9036	tar	0.10	original
gb	0.2196	0.8857	multi	0.10	original
gnb	0.6222	0.5571	tar	0.05	nuevo
hgb	0.2027	0.7964	tar	-	original
knn	0.1450	0.8857	multi	0.25	rápido
mlp	0.1695	0.7464	tar	0.05	o(c)
pac	0.2048	0.7964	tar	0.05	original
rf	0.1939	0.9036	tar	0.10	nuevo
ridge	0.3347	0.8821	multi	0.10	o(c)
sgd	0.2115	0.7929	multi	0.05	original
xgb	0.2365	0.9107	tar	0.25	nuevo
xgbrf	0.2471	0.9143	multi	0.10	original

Tabla 4.27: Cien corridas con combinación, distintos conjuntos de características y con matrices sintéticas

Capítulo 5

Conclusiones y Trabajo Futuro

Este proyecto abordó el problema de selección automática de algoritmos de SpTrSv en *GPU* mediante aprendizaje automático, con el objetivo de minimizar el tiempo total de ejecución considerando preprocesamiento, predicción y ejecución del algoritmo. El trabajo presenta resultados prometedores, donde las distintas metodologías de entrenamiento resultan en modelos muy superiores a las líneas base planteadas.

5.1. Logros y contribuciones

Se consideraron nuevas características a evaluar de las matrices y se plantearon conjuntos distintos buscando balancear el costo de cálculo y expresividad, generando distintas entradas para los modelos. Por otro lado, se evaluaron diferentes técnicas de entrenamiento obteniendo modelos que buscan minimizar el tiempo agregado al predecir erróneamente.

De esta forma se armaron las bases para el sistema integral de selección de algoritmos.

5.1.1. Evaluación de modelos y características utilizadas

Se evaluaron 17 modelos de distintas familias (árboles de decisión, boosting, métodos lineales, entre otros) en tres escenarios, se utilizo Optuna para optimizar sus hiperparámetros y con distintos objetivos de optimización. Obteniendo como mejores resultados los siguientes:

- **Una corrida:** TAR de 0.0396 con Gradient Boosting sin matrices sintéticas (tabla 4.11), y 0.0580 con XGBoost Random Forest incluyendo matrices sintéticas (tabla 4.25)

- **Diez corridas:** TAR de 0.0766 con XGBoost sin matrices sintéticas (tabla 4.12), mejorando a 0.0715 con Perceptrón Multicapa incluyendo matrices sintéticas (tabla 4.26)

- **Cien corridas:** TAR de 0.1500 con Decision Tree sin matrices sintéticas (tabla 4.20), mejorando a 0.1450 con K-Nearest Neighbours usando el conjunto rápido de características e incluyendo matrices sintéticas (tabla 4.27)

Es interesante volver a destacar la interpretación de TAR. El valor de 0.0396 obtenido significa que las predicciones solo significan en un 4 % de tiempo extra del tiempo óptimo.

Se desarrolló la técnica de combinación de algoritmos que permite etiquetar matrices donde dos algoritmos tienen tiempos muy similares como una clase única. Esta técnica obtuvo mejoras en varios modelos, especialmente cuando se utilizaron márgenes de 0.05 o 0.10, permitiendo que los modelos no penalicen fuertemente predicciones entre algoritmos equivalentes en rendimiento.

Todos los modelos entrenados superaron significativamente el TAR esperado de elegir al azar. Por ejemplo, en el escenario de una corrida el TARE era 8.72 mientras el mejor modelo obtuvo 0.0396, y en diez corridas el TARE era 2.27 mientras el mejor modelo obtuvo 0.0715. Esto muestra que los modelos aprenden patrones relevantes para la selección de algoritmos.

A su vez, se desarrollaron tres nuevas características que capturan la distribución espacial de los elementos distintos a cero. Se definieron cuatro conjuntos de características considerando el tiempo de cálculo y se obtuvieron resultados competitivos, principalmente en 100 corridas con el conjunto completo.

También, el conjunto rápido, el cual combina las características de las bandas con las características obtenibles en tiempo constante, fue el vencedor al entrenar con matrices sintéticas, lo que resulta muy prometedor para el sistema integral.

5.1.2. Generación de matrices sintéticas

Una de las principales contribuciones del trabajo es el sistema de generación de matrices. Se implementó un flujo completo que genera matrices a partir de las matrices reales.

Se generaron 3500 matrices sintéticas que fueron utilizadas para expandir el conjunto de entrenamiento. La validación estadística confirmó que las matrices sintéticas mantienen ciertas propiedades relevantes:

- Las distribuciones de densidad coinciden con las matrices reales

- El patrón de mejor algoritmo se mantiene consistente: nan domina en pocas corridas y multi se vuelve dominante conforme aumentan las corridas

- La validación visual durante el entrenamiento mostró que la GAN efectivamente aprendió patrones estructurales como diagonales y distribuciones típicas de matrices dispersas

Por último, se validó que expandir el conjunto de entrenamiento con estas matrices resultó beneficioso, al punto de obtener los mejores resultados del trabajo en los escenarios de 10 y 100 corridas. Esto se vincula directamente con que muchos modelos de aprendizaje automático, requieren más datos para poder obtener buenos resultados, como el Perceptrón Multicapa.

Un detalle no menor, es que si las matrices sintéticas no replicaran los patrones presentes en las reales, al entrenar con ellas y validar en matrices reales se esperarían resultados muy malos. Al no estar sucediendo esto, se puede concluir que la generación de las matrices es apropiada para el problema, aunque puede ser mejorable.

5.2. Trabajo Futuro

A futuro, interesa implementar y evaluar los tiempos completos del proceso, considerando de forma explícita los tiempos de procesamiento de cada característica y de predicción, y ponderar cuánto tiempo dedicar a cada sección del proceso sin perder eficiencia.

El siguiente paso lógico es la implementación del sistema completo en la forma de librería, que permita al usuario ingresar la matriz y los vectores para los cuales quiere realizar la operación y que esta se encargue de seleccionar y ejecutar automáticamente el algoritmo.

A su vez, una biblioteca permitiría evaluar la vialidad y practicidad del sistema para ser integrado en aplicaciones reales de computación científica.

5.2.1. Evaluación de modelos y características

En futuros trabajos, un punto de mejora posible sería cambiar el proceso de evaluación de los modelos cuando estos son entrenados con combinación de algoritmos. Se podrían implementar heurísticas que decidan cuándo el modelo etiqueta una matriz con dos algoritmos. Esta puede utilizar la frecuencia de los mejores algoritmos para definir cuál utilizar o el TAR esperado de elegir siempre cada algoritmo, eligiendo aquel con menor TAR.

Las características utilizadas pueden ser mejoradas ampliamente. Interesa seguir evaluando nuevas características que expresen mejor las dependencias entre filas, los patrones de bloques densos o las estructuras de sub-diagonales. También se podrían evaluar otros criterios para la definición de los conjuntos de características, por ejemplo medir los tiempos que requiere calcular las características y cuantificarlas directamente con su expresividad o utilidad para los modelos, y en base a esos valores definir conjuntos óptimos.

5.2.2. Generación de matrices

Un trabajo futuro sería evaluar una GAN entrenada con imágenes en rangos mayores a 255. Con el hardware disponible, se generaban y evaluaban alrededor de 3 matrices por minuto, por lo que obtener el conjunto de entrenamiento tomó

demasiado tiempo y no se priorizó obtener un nuevo conjunto con una nueva GAN. Este proceso se podría optimizar para mejorar la viabilidad de generar un conjunto aún mayor y que además haga uso de un rango más expresivo que el utilizado (0 a 255).

Otra mejora sería contemplar más canales de información en las imágenes, y no solo representar la densidad en cada pixel, sino que también se podría experimentar con canales que expresen las dependencias verticales, la diagonalidad de los elementos, etc. Una idea posible, es en base a las matrices reales, generar diccionarios de patrones que estas presentan, y utilizar un canal para guardar cuál es el patrón más similar.

Todas estas posibles mejoras, presentan complicaciones en el proceso de downscale y upscale que deben ser atacadas para aprovechar esta mayor información.

También se pueden explorar distintas arquitecturas para la GAN, evaluando si alguna se adecúa más a los patrones presentados en las matrices y en la cantidad de matrices reales disponibles.

Una limitación que se debería atacar para poder generar matrices más realistas, es el tamaño máximo que se puede generar. Con el código realizado y el hardware disponible, intentar aumentar una matriz a tamaños superiores a $100,000 \times 100,000$ agotaba la memoria disponible. Esto genera que el tamaño promedio de las matrices reales sea mucho más alto, dado que las generadas estaban sesgadas hacia tamaños pequeños.

En un principio, parece ser más prometedor implementar procesos posteriores a la generación, donde se rechazan las matrices generadas que difieren demasiado de la realidad o se ajustan por medio de transformaciones para que cuenten con características deseadas, como una cierta profundidad en el grafo de dependencias.

Al ser un proceso inherentemente aleatorio la generación por medio de GANs, puede resultar costoso y muy complejo obtener un Generador que solo genere matrices realistas, por lo que descartar aquellas que difieren demasiado o transformarlas para que contengan ciertos patrones, puede resultar más sencillo.

Por último, interesa expandir el enfoque de todo el trabajo a otras áreas, reutilizando el sistema de generación de matrices o las metodologías de entrenamiento de los modelos para seleccionar automáticamente algoritmos de SpMV o de factorización de matrices dispersas. Existen muchas áreas de la computación donde la selección de algoritmos es crítica, y se podría evaluar si las técnicas como la combinación de algoritmos para el etiquetado de los datos de entrenamiento resulta útil.

Bibliografía

[1] Gene H. Golub and Charles F. van Loan. *Matrix Computations*. JHU Press, fourth edition, 2013. ISBN 1421407949 9781421407944. URL http://www.cs.cornell.edu/cv/GVL4/golubandvanloan.htm.

[2] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2012. ISBN 9780123914187.

[3] Ernesto Dufrechou and Pablo Ezzatti. Using analysis information in the synchronization-free GPU solution of sparse triangular systems. *Concurrency and Computation: Practice and Experience*, 32(10), aug 2019. doi: 10.1002/cpe.5499. URL https://doi.org/10.1002%2Fcpe.5499.

[4] Jennifer Scott and Miroslav Tuma. *Algorithms for sparse linear systems*. Springer Nature, 2023.

[5] Daniel Langr and Pavel Tvrdik. Evaluation criteria for sparse matrix storage formats. *IEEE Trans. Parallel Distrib. Syst.*, 27(2):428–440, February 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2401575. URL https://doi.org/10.1109/TPDS.2015.2401575.

[6] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587448. doi: 10.1145/1654059.1654078. URL https://doi.org/10.1145/1654059.1654078.

[7] Duane Merrill and Michael Garland. Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340922. doi: 10.1145/2851141.2851190. URL https://doi.org/10.1145/2851141.2851190.

[8] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011. ISSN 0098-

3500. doi: 10.1145/2049662.2049663. URL https://doi.org/10.1145/2049662.2049663.

[9] Israt Nisa, Charles Siegel, Aravind Sukumaran Rajam, Abhinav Vishnu, and P. Sadayappan. Effective machine learning based format selection and performance modeling for spmv on gpus. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1056–1065, 2018. doi: 10.1109/IPDPSW.2018.00164.

[10] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939785. URL https://doi.org/10.1145/2939672.2939785.

[11] Shizhao Chen, Jianbin Fang, Chuanfu Xu, and Zheng Wang. Adaptive hybrid storage format for sparse matrix–vector multiplication on multi-core simd cpus. *Applied Sciences*, 12(19), 2022. ISSN 2076-3417. doi: 10.3390/app12199812. URL https://www.mdpi.com/2076-3417/12/19/9812.

[12] Ron Kohavi and George H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1):273–324, 1997. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(97)00043-X. URL https://www.sciencedirect.com/science/article/pii/S000437029700043X. Relevance.

[13] Hang Cui, Shoichi Hirasawa, Hiroyuki Takizawa, and Hiroaki Kobayashi. A code selection mechanism using deep learning. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pages 385–392, 2016. doi: 10.1109/MCSoC.2016.46.

[14] Hang Cui, Shoichi Hirasawa, Hiroaki Kobayashi, and Hiroyuki Takizawa. A machine learning-based approach for selecting spmv kernels and matrix storage formats. *IEICE Transactions on Information and Systems*, E101.D: 2307–2314, 09 2018. doi: 10.1587/transinf.2017EDP7176.

[15] Reo Furuhata, Minglu Zhao, Mulya Agung, Ryusuke Egawa, and Hiroyuki Takizawa. Improving the accuracy in spmv implementation selection with machine learning. In *2020 Eighth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 172–177, 2020. doi: 10.1109/CANDARW51189.2020.00043.

[16] Weijie Zhou, Yue Zhao, Xipeng Shen, and Wang Chen. Enabling runtime spmv format selection through an overhead conscious method. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):80–93, 2020. doi: 10.1109/TPDS.2019.2932931.

[17] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014.

[18] Ernesto Dufrechou, Pablo Ezzatti, Manuel Freire, and Enrique S. Quintana-Ortí. Machine learning for optimal selection of sparse triangular system solvers on gpus. *Journal of Parallel and Distributed Computing*, 158: 47–55, 2021. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2021.07.013. URL https://www.sciencedirect.com/science/article/pii/S0743731521001593.

[19] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521, jan 2005. doi: 10.1088/1742-6596/16/1/071. URL https://dx.doi.org/10.1088/1742-6596/16/1/071.

[20] Mina Ashoury, Mohammad Loni, Farshad Khunjush, and Masoud Daneshtalab. Auto-spmv: Automated optimizing spmv kernels on gpu, 2023. URL https://arxiv.org/abs/2302.05662.

[21] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[22] Zhengding Hu, Jingwei Sun, Zhongyang Li, and Guangzhong Sun. Agsptrsv: An automatic framework to optimize sparse triangular solve on gpus. *ACM Trans. Archit. Code Optim.*, June 2024. ISSN 1544-3566. doi: 10.1145/3674911. URL https://doi.org/10.1145/3674911. Just Accepted.

[23] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014. URL https://arxiv.org/abs/1406.2661.

[24] Cade Dembski, Artemis Spyrou, Sean Liddick, Michelle Kuchera, and Raghu Ramanujan. Using generative adversarial networks to biaxially unfold beta-oslo matrices. In *APS Division of Nuclear Physics Meeting Abstracts*, 2021.

[25] Zehao Yuan, Xuanyan Chen, Biyu Chen, Yubo Luo, Yu Zhang, Wenxin Teng, and Chao Zhang. Generating large-scale origin–destination matrix via progressive growing generative adversarial networks model. *ISPRS International Journal of Geo-Information*, 14(4):172, 2025.

[26] Ernesto Dufrechou, Pablo Ezzatti, and Enrique S Quintana-Ortí. Selecting optimal spmv realizations for gpus via machine learning. *The International Journal of High Performance Computing Applications*, 35(3):254–267, 2021. doi: 10.1177/1094342021990738. URL https://doi.org/10.1177/1094342021990738.

[27] Xinzhe Wu, Serge G. Petiton, and Yutong Lu. A parallel generator of non-hermitian matrices computed from given spectra. *Concurrency and Computation: Practice and Experience*, 2021. doi: 10.1002/cpe.5710.

[28] Massimiliano Fasi and Nicholas J. Higham. Generating extreme-scale matrices with specified singular values or condition numbers. *SIAM Journal on Scientific Computing*, 43(1):A663–A684, 2021. doi: 10.1137/20m1327938.

[29] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer Publishing Company, Incorporated, 2014. ISBN 1461471370.

[30] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL http://www.deeplearningbook.org. Book in preparation for MIT Press.

[31] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997. ISSN 0022-0000. doi: https://doi.org/10.1006/jcss.1997.1504. URL https://www.sciencedirect.com/science/article/pii/S002200009791504X.

[32] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

[33] Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. Catboost: gradient boosting with categorical features support, 2018. URL https://arxiv.org/abs/1810.11363.

[34] Vincent Charles, Tatiana Gherman, and Juan Carlos Paliza. *The Gini Index: A Modern Measure of Inequality*, volume None of *Springer Books*. Springer, none edition, October 2022. doi: 10.1007/978-3-030-84535-3_3. URL https://ideas.repec.org/h/spr/sprchp/978-3-030-84535-3_3.html.

[35] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016. URL https://arxiv.org/abs/1511.06434.

[36] Feng Zhang, Jiya Su, Weifeng Liu, Bingsheng He, Ruofan Wu, Xiaoyong Du, and Rujia Wang. Yuenyeungsptrsv: A thread-level and warp-level fusion synchronization-free sparse triangular solve. *IEEE Transactions on Parallel and Distributed Systems*, 32(9):2321–2337, 2021. doi: 10.1109/TPDS.2021.3066635.

[37] Ernesto Dufrechou and Pablo Ezzatti. Solving sparse triangular linear systems in modern gpus: A synchronization-free algorithm. In Ivan Merelli, Pietro Liò, and Igor V. Kotenko, editors, *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018*, pages 196–203. IEEE Computer Society, 2018. doi: 10.1109/PDP2018.2018.00034. URL https://doi.org/10.1109/PDP2018.2018.00034.