UNIVERSIDAD AUTÓNOMA DE SINALOA

FACULTAD DE INFORMÁTICA CULIACÁN Y
FACULTAD DE CIENCIAS DE LA TIERRA Y EL ESPACIO
POSGRADO EN CIENCIAS DE LA INFORMACIÓN

COMPILADOR MULTI-DESTINO PARA EL DESPLIEGUE DE MODELOS PREDICTIVOS

TESIS

Que como requisito para obtener el grado de
DOCTOR EN CIENCIAS DE LA INFORMACIÓN

Presenta
OSCAR JESÚS CASTRO LÓPEZ

DIRECTOR DE TESIS:
DR. INÉS FERNANDO VEGA LÓPEZ

Culiacán, Sinaloa, México. Febrero de 2020

Le dedico esta tesis a mi familia. A mis padres y en especial a mi esposa Betty y a mi hija Vania.

AGRADECIMIENTOS

Agradezco principalmente a mi asesor de tesis el Dr. Inés Fernando Vega López. Su guía me ha servido para crecer tanto profesionalmente como personalmente. Agradezco su tiempo, su dedicación y especialmente los conocimientos transmitidos.

Agradezco a mis maestros y compañeros que me han ayudado durante esta etapa de mi vida. También agradezco a todo el personal del Posgrado en Ciencias de la Información por su apoyo. Finalmente, deseo agradecer a la Universidad Autónoma de Sinaloa, la Facultad de Informática de Culiacán, el Parque de Innovación Tecnológica y al Consejo Nacional de Ciencia y Tecnología por el apoyo brindado.

ÍNDICE GENERAL

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

RESUMEN

La disponibilidad de datos y el poder de cómputo en la actualidad han permitido un gran crecimiento en la investigación y la práctica de *Machine Learning* (ML). En este crecimiento, se han desarrollado muchas herramientas para construir modelos predictivos utilizando técnicas de *Machine Learning*. Los científicos de datos utilizan estas herramientas para la creación rápida de modelos y la creación de prototipos. Pero obtener valor real de los modelos predictivos requiere su despliegue en entornos de producción. En producción, es en donde las predicciones generadas por el modelo sobre los datos entrantes se integran con productos y servicios para obtener un valor agregado. Un proceso de implementación repetible, rápido y confiable es vital para los flujos de trabajo de *Machine Learning* exitosos.

Los científicos de datos trabajan en su propio entorno con herramientas especializadas. Los científicos de datos persiguen el objetivo específico de construir el mejor modelo predictivo posible a partir de un conjunto de datos determinado y un objetivo. El entorno de producción sigue un conjunto de objetivos completamente diferente, y debe cumplir con varias especificaciones de diseño de software. Mover un modelo predictivo del entorno de modelado a producción (un entorno más restrictivo) para una ejecución eficiente es una tarea difícil. Para empeorar las cosas, la precisión de los modelos predictivos decaen muy rápido. Por lo tanto, la tareas de construcción y despliegue de modelos predictivos se realizan ciclicamente. La traducción manual de modelos predictivos creados utilizando lenguajes de análisis de datos como R o Python a un lenguaje de software utilizado en producción, por ejemplo, C ++ o Java es una tarea lenta y propensa a errores que requiere profesionales altamente especializados. Como opción, es posible conectar directamente el software de producción a herramientas de modelado y, por lo tanto, reducir el tiempo de implementación. Sin embargo, este enfoque es ineficiente y tiene una deuda técnica oculta que aumenta significativamente con el tiempo.

Nuestra propuesta se basa en los siguientes hechos. Cada herramienta analítica tiene

su propia representación interna que captura todas las características de un modelo. Estas características se pueden expresar de manera formal. Además, existen diferentes estándares para describir modelos predictivos, como el *Predictive Model Markup Language* (PMML), el *Portable Format for Analytics* (PFA) y el *Open Neural Network Exchange* (ONNX). Por lo tanto, para facilitar la implementación, hemos diseñado y desarrollado un compilador de propósito especial para automatizar la traducción de modelos predictivos de su descripción formal a código fuente en un lenguaje de programación. El compilador propuesto se compone de dos módulos principales: el *front-end* y el *back-end*. Además, hemos desarrollado una representación intermedia (IR de *Intermediate Representation*) que sirve de puente entre los dos módulos principales. El *front-end* traduce descripciones de modelos predictivos a la IR. El *back-end* traduce una IR a un lenguaje de programación objetivo específico. También hemos desarrollado un conjunto de plantillas para generar el IR que implementa la función de predicción de modelos predictivos. El diseño basado en representaciones intermedias del compilador admite una arquitectura dinámica que permite muchos tipos diferentes de modelos como entradas y muchos lenguajes de programación de destino como salidas. Definimos varios bloques de construcción, tales como operaciones de álgebra lineal y funciones específicas de aprendizaje automático para tener una IR ligero.

Al compilar un modelo predictivo, nuestro objetivo es la eficiencia del tiempo de ejecución del modelo implementado en la arquitectura de la computadora donde se ejecuta el sistema en producción. Por lo tanto, no solo podemos implementar en varios lenguajes de programación como C, Java y SQL (a través de funciones definidas por el usuario), sino que también explotamos características específicas de la arquitectura, como múltiples núcleos y la disponibilidad de tarjetas de procesamiento gráfico. Además, la modularidad en nuestro compilador permite una extensión fácil, se pueden agregar nuevas descripciones de modelos predictivos y nuevos idiomas de destino según sea necesario.

A través de la implementación de un compilador, demostramos la viabilidad de nuestra propuesta. Con los resultados de una evaluación experimental, mostramos que nuestro

código generado es efectivo y puede ejecutarse eficientemente en diferentes lenguajes y arquitecturas de destino. Además, demostramos que el código generado puede superar a otras herramientas analíticas en la mayoría de los casos.

CAPÍTULO 1
INTRODUCCIÓN

Existe un gran volumen de datos generados por los sistemas de información. Según un informe de KPMG, la cantidad de información digital aumentará diez veces cada cinco años [1]. La recopilación de datos se está habilitando porque el costo de almacenamiento ha disminuido, la proliferación de dispositivos conectados a Internet (IoT) que generan datos y la mejora en las redes de comunicación que permite un intercambio de información más rápido. El enorme crecimiento y diversidad de conjuntos de datos generados por empresas privadas, gobiernos, sensores, transacciones y redes sociales representa una gran oportunidad para extraer información relevante [2].

Debido a la disponibilidad de datos y poder de cómputo, estamos viendo un creciente interés en el análisis de datos, especialmente en el análisis predictivo. Con el análisis predictivo analizamos eventos pasados (datos) para tratar de predecir eventos futuros. Las técnicas estadísticas y/o de aprendizaje automático (*Machine Learning*) se utilizan para construir modelos predictivos. Un modelo predictivo es una función matemática que describe la relación entre un atributo (variable dependiente) de un objeto en una muestra con respecto a uno o más atributos (variables independientes) del mismo objeto.

Dentro del alcance de esta disertación, el término aprendizaje automático se utiliza para referirse a técnicas de aprendizaje automático supervisado. En pocas palabras, el aprendizaje supervisado se refiere al proceso de inferir una función a partir de ejemplos con entradas y salidas conocidas. Dicha función recibe un conjunto de parámetros (datos de entrada) y devuelve un valor de salida (la predicción). Dependiendo del tipo de técnica de aprendizaje automático, los parámetros de entrada y el valor de salida pueden ser discretos o continuos. Si el resultado de un modelo es un valor categórico, la tarea de predicción se conoce como clasificación; de lo contrario, si el resultado es un valor continuo, la tarea de predicción se

conoce como regresión.

Debido a su potencial, los modelos predictivos ofrecen el mayor valor entre las disciplinas analíticas. Una encuesta realizada por el MIT e IBM descubrió que las empresas de mayor rendimiento tienen más probabilidades de utilizar métodos analíticos, y ven el análisis predictivo como un atributo que los distingue de sus competidores [3]. Otro estudio del MIT [4], que encuestó a 179 compañías, muestra que tomar decisiones basadas en análisis de datos produce entre un cinco y un seis por ciento más de productividad de lo que se esperaría de tecnologías o inversiones sin el uso de análisis de datos. Ese aumento del cinco por ciento ofrece a las empresas una gran ventaja frente a sus competidores en la mayoría de las industrias. Estos estudios confirman la importancia de la adopción del análisis predictivo para mejorar la toma de decisiones, descubrir información que no está a la vista, desencadenar acciones automáticamente y optimizar sus procesos.

Las tareas relacionadas con el análisis de datos son realizadas por científicos de datos, una rol estrechamente vinculado con las estadísticas, las matemáticas y el procesamiento de datos. Los científicos de datos también requieren conocimientos del dominio de los datos y habilidades analíticas. La tarea de construir modelos predictivos se conoce como modelado predictivo, es un proceso complejo en el que los científicos de datos hacen uso de diferentes disciplinas como las estadísticas, el aprendizaje automático y la informática. Estas disciplinas principales se utilizan para analizar conjuntos de datos, también se utilizan otras sub-disciplinas como la minería de datos, la optimización y el reconocimiento de patrones. El modelado predictivo es un proceso creativo, guiado por analistas, donde se utilizan muchas herramientas especializadas y repositorios de datos. La interacción de varios programas, herramientas y *scripts* aumenta la complejidad de la tarea de modelado predictivo. Un flujo de trabajo de modelado analítico típico es el siguiente.

1. Crear conjuntos de datos para analizar mediante la recopilación de datos de diferentes fuentes (selección de registros, agrupación y transformación de datos).

2. Realizar un análisis exploratorio de los datos (por ejemplo: gráficos, agrupamiento,

relaciones entre variables y valor de información).

3. Crear modelos predictivos (con la ayuda de herramientas especializadas como R, Phyton, SPSS o SAS).

4. Validar la precisión de los modelos y, si no se cumplen los objetivos, repetir los pasos anteriores.

5. Implementar el modelo en un entorno de producción de software para generar predicciones sobre nuevos datos.

6. Monitorear y actualizar modelos.

El flujo de trabajo del modelado predictivo requiere varias iteraciones antes de obtener resultados que cumplan un objetivo. Los modelos predictivos que se ejecutan en producción requieren monitoreo y actualizaciones constantes. Esto se debe a que su eficiencia de predicción decae o debido a que los datos cambian con el tiempo (*data drift*). Las características de los datos mutan de manera impredecible con el tiempo, esto se conoce como deriva de datos. Por lo tanto, monitorear y actualizar modelos predictivos es un proceso iterativo de repetición constante.

Después de la validación, los modelos se implementan en aplicaciones del mundo real (entorno de producción) donde se puede obtener valor con las predicciones que se generan en los nuevos datos de entrada. Dentro del contexto de esta investigación, un entorno de producción es donde el software se pone realmente en funcionamiento para su uso previsto por los usuarios finales. El proceso de hacer una predicción basada en datos de entrada usando un modelo predictivo (construido con información histórica) se llama *scoring*, y el resultado se conoce como *score*. Dependiendo del contexto, la puntuación también se conoce como predicción o inferencia.

Los modelos predictivos se entregan a los ingenieros de software, quienes se encargan de integrar estos modelos como parte de un servicio o producto de software, esta tarea se

trata como cualquier otra tarea de ingeniería de software. Sin embargo, la implementación de modelos predictivos no es una tarea de ingeniería de software convencional. Por un lado, el lenguaje técnico entre el modelado predictivo y la ingeniería de software es muy distinta. Adicionalmente, los modelos predictivos tienen una alta complejidad. Codificar la lógica y las matemáticas detrás de los modelos es una tarea difícil incluso para los ingenieros de software más experimentados [5].

Formalmente, podemos definir la tarea de implementación de modelos predictivos de la siguiente manera.

Dado un modelo predictivo A, que es una función matemática generada por computadora expresada en el lenguaje L, queremos generar A' expresado en un lenguaje de programación de computadora L', en donde A' es semánticamente equivalente a A.

Intuitivamente, la implementación es el proceso general de llevar un modelo predictivo a un entorno operativo específico donde está disponible para su uso (software). La implementación también se conoce como operacionalización o análisis operativo. El servicio de modelo también es un término comúnmente utilizado, aunque se refiere a una arquitectura cliente-servidor donde una aplicación cliente envía datos y una solicitud de predicción a un servidor donde se aloja un modelo predictivo. El servidor utiliza el modelo para generar predicciones y las envía de vuelta al cliente.

Un modelo predictivo es una especificación formal de una función que debe integrarse con el software en producción. A partir de dicha especificación formal (que es generada por computadora), es posible transformarla algorítmicamente (es decir, compilar) a un lenguaje de programación de alto nivel. Proponemos automatizar la tarea de implementación de modelos predictivos con un compilador, que puede facilitar la integración de modelos predictivos con el software en producción. Se puede usar una descripción formal de un modelo predictivo como entrada para el compilador. Como salida, se genera el código fuente equivalente del modelo predictivo en un lenguaje de programación, lo que permite una fluida integración con el software de producción. El compilador puede tener un enfoque

multi-destino para poder generar código fuente para diferentes lenguajes de programación de acuerdo con el entorno de producción.

El objetivo principal de esta disertación es la automatización de la tarea de implementación del modelo predictivo inferido por técnicas de aprendizaje automático (*machine learning*) y su ejecución eficiente en el software de producción. Para lograr este objetivo, hemos desarrollado un compilador de multi-destino para el despliegue de modelos predictivos, utilizando una descripción formal del modelo como entrada. *Multi-destino* significa que el compilador debe poder generar código para diferentes lenguajes de programación o arquitecturas de computadora. La estructura central de los algoritmos de predicción de los modelos predictivos, en general, se puede expresar formalmente en términos de álgebra lineal. Por lo tanto, es una descripción matemática y se puede definir una representación intermedia para facilitar la traducción a varios lenguajes de programación destino. El código resultante del compilador es semánticamente equivalente al modelo predictivo de entrada. Esta es una forma efectiva de implementar modelos predictivos por las siguientes razones.

- Eficiencia de implementación; Reduce la cantidad de recursos necesarios para implementar un modelo. Por recursos nos referimos a las horas-hombre dedicadas a codificar el modelo predictivo en un lenguaje de programación. Por lo tanto, el tiempo y el costo de la implementación se reducen drásticamente.

- Libre de errores; Los errores inducidos por humanos se eliminan ya que no hay traducción manual.

- Eficiencia de ejecución; El código fuente generado está listo para ser compilado para una máquina destino específica. El compilador también puede generar código optimizado dependiendo de la arquitectura de la máquina destino.

Mientras que la traducción de una descripción formal de un modelo predictivo al código fuente en un lenguaje de programación de computadora se puede realizar manualmente, es más práctico usar un compilador. A través de una serie de procedimientos automatiza-

dos, el compilador puede generar el código fuente requerido para implementar el modelo. Primero, el modelo predictivo descrito en un lenguaje estándar debe analizarse y traducirse a una Representación Intermedia (*Intermediate Representation*, IR) desarrollada para el compilador. Luego, la IR se procesa para generar código para diferentes lenguajes de programación (enfoque de multi-destino).

La principal contribución de esta disertación es un compilador para traducir modelos predictivos a código fuente en un lenguaje de programación destino para automatizar la tarea de implementación. La salida del compilador es el código fuente equivalente en la arquitectura de destino seleccionada y/o el lenguaje de programación que contiene el algoritmo de predicción que se puede utilizar para implementar el modelo. Las contribuciones clave de este trabajo se enumeran a continuación.

- Un enfoque novedoso para automatizar la implementación de modelos predictivos mediante la generación de código fuente eficiente en lenguajes de programación de software.
- El desarrollo de un compilador que traduce modelos predictivos formalmente definidos al código fuente para ejecutarse en diferentes paradigmas de programación como secuencial (CPU), multi-core (CPU) y muchos-core (GPU).
- Una evaluación empírica que demuestra que el código generado (en diferentes entornos y objetivos) es eficiente.

El resto de la disertación se organiza de la siguiente manera. El capítulo 2 proporciona información preliminar y una revisión de la literatura del trabajo relacionado. La revisión de la literatura incluye una breve introducción a los conceptos básicos en el área de la ciencia de datos, el modelado predictivo y una revisión de los enfoques actuales para el despliegue del modelo predictivo. En el Capítulo 3, se presenta una descripción general del compilador multi-destino propuesto para implementar modelos predictivos. Además, también se presentan las plantillas de representación intermedia de los modelos predictivos.

El Capítulo 4 describe el diseño y la implementación del compilador propuesto. El Capítulo 5 presenta resultados experimentales donde comparamos la eficiencia de la ejecución del código fuente generado por el compilador propuesto. Finalmente, el Capítulo 6 resume las contribuciones de esta disertación, discutiendo las limitaciones actuales de nuestro enfoque y trabajo futuro.

CAPÍTULO 2
TRABAJO RELACIONADO

En este capítulo, se describen los trabajos y conceptos relacionado con esta disertación. Primero, se proporciona información preliminar para entrar en contexto sobre la implementación de modelos predictivos. Posteriormente, se revisan aspectos relevantes del modelado predictivo y la implementación. Se describe un flujo de trabajo para construir modelos predictivos y los enfoques de implementación actuales. Al final de este capítulo, revisamos algunas oportunidades de investigación relacionadas con la implementación de modelos predictivos.

2.1 Preliminares

El análisis de datos y el aprendizaje automático (*machine learning*) son términos muy populares hoy en día, aunque han existido durante bastante tiempo con un nombre diferente. A finales de los 90, la minería de datos se utilizó para describir el proceso de descubrir patrones y conocimientos interesantes a partir de grandes cantidades de datos [6]. Otro término popular fue Knowledge Discovery in Databases (KDD), que es el proceso de encontrar conocimiento en datos. La minería de datos es un área interdisciplinaria que incorpora muchas técnicas de otros dominios, como estadística, aprendizaje automático, reconocimiento de patrones, bases de datos, recuperación de información, visualización, algoritmos, cómputo de alto rendimiento y muchos dominios de aplicación específicos. Las tareas de minería de datos se pueden clasificar en dos categorías principales: 1) análisis descriptivo, el cual caracteriza las propiedades de los datos en un conjunto de datos objetivo, y 2) análisis predictivo, realiza inducciones sobre los datos actuales para hacer predicciones.

En los últimos años, el uso del término minería de datos ha disminuido, y el término

ciencia de datos se ha vuelto más popular. No existe una definición unificada para el término ciencia de datos, pero la siguiente definición proporcionada por Dhar se usa comúnmente: *"La ciencia de datos es un estudio sistemático de métodos, procesos y sistemas científicos para extraer conocimiento o ideas de los datos en varias formas"*. [7]. La ciencia de datos también se describe como la intersección de la estadística y la informática, y como un campo interdisciplinario que utiliza técnicas y teorías dentro de las áreas de matemáticas, estadística, ciencias de la información (bases de datos), informática e inteligencia artificial. Las definiciones de ciencia de datos y minería de datos son muy similares. Ambos son campos interdisciplinarios que toman prestadas técnicas de otros dominios y algunas de sus actividades pueden mapearse.

A lo largo del crecimiento del término ciencia de datos, ha surgido un nuevo rol conocido como científico de datos. Kim et al. realizó un estudio en el que organizaron las actividades de los científicos de datos en tres categorías: recopilación de datos, análisis de datos y uso y difusión de datos [8]. Cada actividad tiene sus sub-actividades. La Tabla 2.1 resume estas actividades y sub-actividades.

Dentro del contexto de esta disertación, estamos interesados en el despliegue (*operacionalización*) de modelos predictivos. Primero, es importante revisar el proceso de construcción de un modelo predictivo. Este proceso se conoce comúnmente como modelado predictivo. El objetivo del modelado predictivo es construir un modelo donde el valor de una variable se pueda predecir a partir de los valores de otras variables. Dicho modelo se denomina modelo de *machine learning* o modelo predictivo, que se puede definir como una función matemática que describe la relación entre una clase objetivo y una o más variables (características) independientes.

El modelado predictivo se basa en técnicas de aprendizaje automático. El aprendizaje automático es una forma de estadística aplicada con el uso de computadoras para estimar estadísticamente (calcular) funciones complicadas y un menor énfasis en probar intervalos de confianza en torno a estas funciones [9]. El aprendizaje automático es el estudio de

Actividades	Sub-actividades
Recopilacion de datos	• Ingeniería de una plataforma de datos: construcción de un sistema para recopilar datos de múltiples fuentes continuamente. • Inyección de telemetría: insertando código de instrumentación para recopilar perfiles de ejecución y uso de software. • Creación de una plataforma de experimentación: creación de capacidad inherente para la experimentación con diseños de software alternativos.
Análisis de datos	• Unión y limpieza de datos: unir datos de múltiples fuentes y tratar con valores faltantes e instrumentación imperfecta. • Muestreo de elementos: selección de un subconjunto de un conjunto de individuos dentro de una población estadística. • Modelado de datos: selección de características, transformación de datos en un nuevo formato y creación de nuevos atributos en un vector de características. • Definición de métricas: definición de métricas que son sensibles para los consumidores de datos. • Construcción de modelos predictivos: construcción de modelos predictivos aplicando aprendizaje automático, minería de datos y estadísticas. • Etiquetado de elementos: definición de etiquetas de clase y escenarios de anomalías de datos. • Prueba de hipótesis: establecer una hipótesis nula y una hipótesis alternativa y estimar el nivel de confianza de rechazar la hipótesis nula utilizando varios métodos estadísticos.
Uso y difusión de datos	• Operacionalización de modelos predictivos: integración de modelos predictivos en productos de software y sistemas invocando modelos correctos en el momento correcto. • Definición de acciones y disparadores: definición de acciones automatizadas y disparadores para diferentes etiquetas de predicciones. • Traducción de ideas y modelos a valores empresariales: explicación del valor de las ideas y modelos predictivos utilizando términos específicos del dominio.

Tabla 2.1: Actividades principales y sub-actividades de científicos de datos (basadas en [8]).

cómo con el uso de las computadoras y sus programas se puede aprender de los datos, para reconocer automáticamente patrones y comportamientos dentro de los datos [6]. Los algoritmos de aprendizaje automático se pueden dividir en las siguientes categorías; aprendizaje supervisado, aprendizaje no supervisado y aprendizaje de refuerzo.

Los modelos predictivos se crean utilizando algoritmos de aprendizaje supervisado. La entrada de los algoritmos de aprendizaje supervisado es un conjunto de datos donde cada elemento está asociado con una etiqueta o variable objetivo. Esto significa que el conjunto

de datos contiene las salidas correctas de una función desconocida para entradas particulares. De manera concisa, un algoritmo de aprendizaje supervisado funciona de la siguiente manera. Un modelo se inicializa con parámetros aleatorios o dados, cada elemento dado del conjunto de datos es procesado (*visto*) por el algoritmo y se produce una salida. Posteriormente, la salida se compara con el resultado conocido (variable de destino). La distancia / diferencia entre la salida y el resultado real permite ajustar los parámetros del modelo para el siguiente elemento. Estos pasos se repiten un número definido de épocas o hasta que se alcanzan ciertas condiciones. El propósito es generar (inferir) la función desconocida o una función que se aproxima a ella.

Formalmente, podemos definir el aprendizaje supervisado de la siguiente manera.

Dado un conjunto de pares de ejemplos conocidos (X, y) *donde hay una función desconocida* $f()$ *que mapea* X *a* y*, queremos inferir la función* $h()$ *que se aproxima a* $f()$.

La función desconocida h se llama hipótesis. El aprendizaje es un desafío porque no podemos saber si alguna h en particular es una buena aproximación de f. Una buena hipótesis generalizará bien; va a predecir ejemplos no vistos correctamente [10]. Algunas técnicas populares de aprendizaje supervisado son: redes neuronales (NN), máquinas de vectores de soporte (SVM), árboles de decisión, regresión lineal y regresión logística.

Los algoritmos de aprendizaje no supervisados aprenden propiedades útiles de la estructura de un conjunto de datos que no tiene etiqueta o variable objetivo. Las técnicas de aprendizaje no supervisadas se utilizan para extraer inferencias de conjuntos de datos sin etiquetas o variables objetivo. Abarcan principalmente técnicas de agrupamiento y es muy utilizado en el análisis exploratorio de datos para encontrar patrones ocultos o elementos de agrupación de un conjunto de datos. Las técnicas de agrupamiento, *agrupan* observaciones de datos calculando cuánto se parecen entre sí. Algunas técnicas de agrupamiento son: agrupamiento K-means, mapas auto-organizados, agrupamiento jerárquico y descenso de gradiente estocástico (SGD).

Dentro de los contextos de aprendizaje automático o modelado predictivo, hay dos etapas principales involucradas; Entrenamiento y predicción. En el entrenamiento, se utiliza un conjunto de datos como entrada para un algoritmo que estima los parámetros de un modelo. El resultado de este proceso es un modelo predictivo. A un modelo predictivo también se le conoce como modelo entrenado (*trained*) o ajustado (*fitted*). Esto se debe a que, independientemente de la técnica de aprendizaje automático utilizada, estamos tratando de *ajustar* el modelo a los datos de entrada. Por ajuste queremos decir que el modelo resultante es confiable y preciso para hacer predicciones. En la predicción, se utilizan nuevos datos como entrada para un modelo previamente entrenado, dicho modelo genera una predicción. A la predicción también se le denomina como *scoring* y su resultado se conoce como predicción, *score* o inferencia. Los atributos de entrada de un proceso de *scoring* pueden ser discretos o continuos según el tipo de modelo predictivo. La salida de un modelo también puede ser discreta o continua; aprender una función de valor discreto se llama clasificación, aprender una función de valor continuo se llama regresión.

Los científicos de datos utilizan una amplia variedad de herramientas de software para procesar datos y construir modelos predictivos. Dichas herramientas se conocen como herramientas de modelado, herramientas analíticas, herramientas estadísticas o marcos de aprendizaje automático (*machine learning frameworks*). Todas estas herramientas ofrecen un conjunto de funcionalidades para realizar tareas de análisis y procesamiento de datos, así como un conjunto de técnicas de modelado analítico. Existen diferentes metodologías para llevar a cabo un proyecto que involucra modelado predictivo, como KDD (del inglés *Knowledge Discovery in Databases*) [11], SEMMA (del inglés *Sample-Explore-Modify-Model-Evaluation* [12] , o el CRISP-DM (del inglés *Cross Industry Standard Process for Data Mining*) [13]. Muchos expertos utilizan el proceso CRISP-DM o un proceso muy similar [14, 15]. Independientemente del proceso utilizado, un proyecto de ciencia de datos comprende varias tareas antes y después de la construcción del modelo.

2.2 Despliegue de modelos predictivos

En comparación con la disponibilidad de herramientas y propuestas para construir modelos predictivos, las propuestas destinadas exclusivamente a realizar la tarea de despliegue son escasas. Ha habido propuestas individuales que abarcan el ciclo completo de construcción y despliegue de modelos predictivos. Este tipo de propuestas, comprenden una herramienta o un conjunto de herramientas diseñadas para realizar todas las tareas relacionadas con el modelado predictivo, como el procesamiento de datos, el entrenamiento de modelos y la implementación de modelos. En la literatura se han hecho algunos intentos para clasificar los enfoques de despliegue. Por ejemplo, Grossman presenta un marco para evaluar la madurez analítica de una organización (APMM por sus siglas en inglés) [16]. El APMM es un proceso modelo para análisis, comparable con el modelo de integración de sistemas modelos de madurez de capacidades (CMMI por sus siglas en inglés) que está dirigido al desarrollo, mantenimiento y operación del software. Grossman enumera los siguientes enfoques para el despliegue de modelos predictivos.

- Uso de herramientas de modelado en línea. La misma aplicación utilizada para la construcción de modelos es utilizada para generar predicciones en producción.

- Uso de herramientas de modelado *offline*. El modelo genera predicciones en modo fuera de línea. Dichas predicciones se almacenan en una base de datos para ser utilizadas posteriormente por el entorno de producción.

- Uso de motores de *scoring*(*scoring engines*). El modelo se exporta desde un entorno de modelado y se importa en el entorno de producción. El modelo se expresa en algún lenguaje o formato estándar, por ejemplo: en PMML (del inglés *Predictive Model Markup Language*)[17] o en PFA (del inglés *Portable Format for Analytics*) [18]. Los modelos son consumidos por un motor de *scoring*, que interpreta el modelo y genera predicciones sobre los datos de entrada.

- Codificación manual. El modelo es codificado manualmente por un equipo de ingenieros de software e integrado directamente en el entorno de producción.

Lee et al. describen tres enfoques principales para realizar la tarea de despliegue [19]. El primer enfoque es implementar modelos en contenedores. Un contenedor es una unidad virtual de software. En este enfoque, el contenedor tiene todo el software y los paquetes necesarios para ejecutar el modelo previamente entrenado. Esto permite utilizar la misma herramienta utilizada para el entrenamiento de modelos en la fase de implementación. El software en producción se comunica con el contenedor, por ejemplo, a través de una llamada a procedimiento remoto (RPC) a un servidor web. El segundo enfoque es importar modelos directamente a las aplicaciones escribiendo la lógica directamente en la aplicación. El tercer enfoque es la propuesta de Lee et al. llamado servicio de modelo de caja blanca, donde las aplicaciones acceden a un servidor de *scoring* en tiempo de ejecución para utilizar modelos predictivos.

Rasiwasia define tres mecanismos populares para llevar modelos predictivos a producción [20]. El primer mecanismo está basado en servicios y utiliza la arquitectura orientada a servicios (SOA) para implementar modelos. Las aplicaciones en producción se comunican con los modelos implementados una SOA a través de un protocolo de comunicación utilizando una red. El segundo mecanismo consiste en compartir el modelo predictivo a través de algún archivo serializado o el código para generar predicciones. De una manera más avanzada, se puede compartir dentro de un contenedor, como un contenedor Docker. El tercer y último mecanismo es el basado en resultados, en el que las predicciones se generan en modo *offline* y los resultados se comparten directamente para ser utilizados por otra aplicación o usuarios finales.

Las categorizaciones anteriores ofrecen una buena visión general de las diferentes propuestas, herramientas o enfoques para el despliegue de modelos predictivos. Los enfoques más relevantes con respecto a esta disertación se agruparon y revisaron de la siguiente manera. En la Sección 2.2.1, se enumeran los trabajos que siguen un enfoque cliente-servidor,

donde el servidor contiene una herramienta de modelado donde se utiliza un modelo previamente entrenado para generar predicciones. La Sección 2.2.2 se centra en los enfoques en los que los modelos se implementan en los sistemas gestores de bases de datos (DBMS). En la Sección 2.2.3, se enumeran otros tipos de implementación que no corresponden a las categorías anteriores.

2.2.1 Enfoque de despliegue cliente-servidor

El enfoque cliente-servidor es una de las opciones de implementación más utilizadas. En este caso, el cliente es el software en producción que, utilizando un protocolo de comunicación, envía una solicitud de predicción al servidor. Esta solicitud contiene el vector de características de un objeto para el que se requiere generar una predicción. El servidor es un software con al menos dos componentes principales: 1) un intermediario con un protocolo de comunicación para definir la forma en que el servidor recibirá las solicitudes, el formato de la solicitud y el formato de respuesta al cliente, 2) un *back-end* de predicción interno, que es un componente de software que aplica un modelo predictivo a los datos entrantes para calcular las predicciones. Internamente, el servidor debe tener un modelo (o un conjunto de modelos) cargado o listo para cargarse bajo pedido. Posteriormente, el *back-end* de predicción aplica el modelo solicitado a los datos entrantes. Finalmente, el servidor debe devolver las predicciones resultantes al cliente a través del protocolo de comunicación definido. La Figura 2.1 muestra un esquema del enfoque cliente-servidor.

El software que funciona como *back-end* para generar predicciones o *scores* se puede seleccionar de diferentes opciones. Una opción popular es usar la misma herramienta de modelado donde se construyó el modelo. Otra opción es utilizar una herramienta de software especial conocida como motor de *scoring*, que carga e interpreta los modelos guardados en un formato definido y genera predicciones sobre los datos entrantes. Para evitar problemas durante la configuración del entorno del servidor, una opción disponible es encapsular una herramienta de modelado o un motor de *scoring* en un contenedor. Un

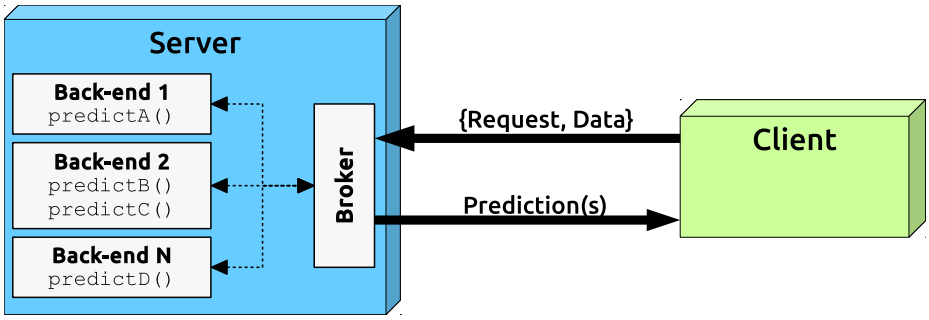


Figura 2.1: Vista gráfica de un modelo predictivo implementado en una arquitectura cliente-servidor. El software en producción es el cliente y envía datos al servidor, que en este caso es un software encapsulado que devuelve predicciones.

contenedor es una unidad virtual de software que empaqueta todo el código y sus dependencias para ejecutar los modelos predictivos (por ejemplo, Docker [21]). Las opciones basadas en la nube también se utilizan como *back-end* de predicción. En este caso, los modelos se crean utilizando una plataforma en la nube y luego se implementan como servicios web. A continuación se ofrece una descripción más detallada de cada variante del enfoque cliente-servidor.

***Pipelines*.** En este enfoque, la conexión se crea directamente entre el software en producción y el software de modelado analítico. Por ejemplo: R, Python (Scikit Learn [22]), Knime, SAS o SPSS. Estas herramientas tienen paquetes o bibliotecas disponibles para exponer el código que se conectará a las aplicaciones en producción. El código expuesto carga el modelo e invoca la función de predicción en los datos entrantes. Este código expuesto también puede tener pasos de procesamiento de datos antes de llamar a la función de predicción. En este enfoque cliente-servidor, en el servidor se ejecuta la misma herramienta donde se construyó el modelo; Se cargan todos los paquetes y dependencias, así como el modelo. El servidor está listo para recibir datos y devolver predicciones utilizando algún protocolo de comunicación como RPC o una API REST (servicio web). Este enfoque permite una implementación rápida y fácil de modelos predictivos. Sin embargo, estas herramientas no fueron diseñadas originalmente para ser utilizadas en ambientes en

producción. Por lo tanto, no son adecuados para un entorno de producción de alta demanda ni se escalan fácilmente.

En general, las herramientas de modelado no son adecuadas para aplicaciones de alto rendimiento o para trabajar en un entorno distribuido. Por lo tanto, se han desarrollado nuevas herramientas de modelado o se han ampliado las existentes para mejorar la ejecución de las tareas de entrenamiento y predicción al aprovechar entornos distribuidos o paralelos. Por ejemplo, SystemML [23] es un lenguaje y un software para construir y ejecutar modelos predictivos en un entorno distribuido en el paradigma de programación MapReduce. SystemML permite expresar algoritmos de aprendizaje automático en un lenguaje de alto nivel que se puede compilar y ejecutar en un entorno MapReduce. En una versión posterior, SystemML evolucionó para poder ejecutar modelos predictivos en Apache Spark [24]. Otra herramienta similar es PredictionIO [25], que permite construir e implementar modelos predictivos como un servicio web utilizando Spark y Hadoop. Las aplicaciones en producción envían datos y una consulta de predicción al servidor PredictionIO. Se envía una consulta de predicción utilizando un kit de desarrollo de software (SDK). Estos SDK son proporcionados por PredictionIO y está disponible para diferentes lenguajes de programación. Después de que el servidor recibe una consulta, el modelo solicitado se ejecuta y las predicciones se devuelven al cliente. PredictionIO sirve como intermediario entre herramientas de modelado y entornos de producción.

Siguiendo la tendencia destinada a acelerar los cálculos utilizando arquitecturas distribuidas, han surgido herramientas como MLlib [26]. MLlib es una biblioteca de aprendizaje automático construida sobre Spark. Se puede usar con interfaces en los lenguajes Scala, Java, R y Python. MLlib proporciona muchos algoritmos populares de aprendizaje automático. Los modelos construidos con MLlib se pueden implementar más tarde dentro de un clúster Spark. Otra herramienta basada en Spark es KeystoneML [27], diseñado para realizar las tareas del ciclo de aprendizaje automático. Con KeystoneML, los modelos pueden construirse definiendo una especificación de alto nivel y pueden ser desplegados en un

entorno Spark.

Las extensiones de herramientas de modelado también son un enfoque explorado. Ma et al. presentan ddR (en inglés *Distributed Data structures in R*), un sistema que permite usar R en diferentes ambientes distribuidos (y paralelos) [28]. En ddR, se proponen primitivas de programación llamadas *dmapply* y se utilizan para ejecutar funciones en estructuras de datos distribuidos. Las primitivas *dmapply* encapsulan diferentes patrones de cómputo que permiten expresar algoritmos de aprendizaje automático y aplicaciones escritas en un enfoque MapReduce. El sistema ddR puede integrarse con diferentes back-end como la herramienta Single-node parallel framework de R, multi-SNOW framework, Spark y HPE Distributed R. En resumen, el sistema ddR extiende el lenguaje R proporcionando una interfaz unificada para el cómputo distribuido. El sistema ddR se puede utilizar para el entrenamiento y el despliegue de modelos predictivos aprendidos por máquina. Kunft et al. proponen ScootR [29], una herramienta para ejecutar scripts de R usando Apache Flink [30] como motor de procesamiento de datos. ScootR le permite a R acceder a los tipos de datos abstractos de Flink. Por lo tanto, las funciones R definidas por el usuario se pueden construir e implementar en un clúster de Flink y ejecutar en paralelo por los nodos del clúster. Aunque ScootR originalmente no se concibió exclusivamente para la implementación de aprendizaje automático, se puede usar como una herramienta de implementación además de Apache Flink.

Aloha es un *framework* presentado por Deak y Morra [31] que permite construir una especificación de tareas de aprendizaje automático utilizando su propio lenguaje específico de dominio (DSL) basado en Scala. Aloha actúa como un puente entre las representaciones de datos y los paquetes de aprendizaje automático (*machine learning*). Este *framework* permite procesar un conjunto de datos y entrenar modelos predictivos. El modelo entrenado y sus metadatos están almacenados en un archivo de especificación que puede usarse más tarde para la predicción. Para implementar un modelo con Aloha, se debe crear un intérprete de modelo, con un auditor que establezca el formato de predicción y la información de

diagnóstico devuelta por el modelo. Aloha intercambia datos mediante el uso de formatos de serialización basados en protocolos, incluidos los *buffers* de protocolo y Avro (un formato de serialización de datos binarios en JSON). Aloha consume datos con esquemas definidos por el usuario.

Talagala et al. presentan Edge Cloud Orchestrator (ECO) [32], una arquitectura para permitir la implementación de modelos predictivos para edge y cloud. ECO está diseñado con una arquitectura distribuida de agente de servidor. El servidor se comunica con diferentes agentes; cada agente gestiona un motor analítico que ejecuta un conjunto de tareas de aprendizaje automático. Actualmente, se admiten tres herramientas de modelado (conocidas como motores analíticos en el diseño de arquitectura ECO): Spark-MLlib [33], Flink [30] y TensorFlow [34]. Cada agente debe comunicarse con su motor y al menos tener comunicación intermitente con el servidor para las actualizaciones del modelo. Los agentes pueden ejecutarse en dispositivos informáticos *edge* o *cloud*. En su propuesta, los agentes trabajan como intermediarios entre las herramientas de modelado, los repositorios de datos y las aplicaciones del usuario final. La implementación de ECO se creó con Java y SQLite, y la comunicación entre agentes y servidores se realiza a través de las API REST.

Lee et al. proponen PRETZEL [19, 35], un software para ML.NET (herramienta de *machine learning* de Microsoft) creado específicamente para el servicio de predicción. En PRETZEL, el despliegue y la entrega de modelos siguen un proceso de dos fases. Una fase *offline* donde los modelos previamente entrenados y los pasos de procesamiento de datos se traducen en unidades lógicas llamadas etapas. Cada etapa se compila en unidades de cómputo. En la fase en línea, cuando se emite una predicción, el tiempo de ejecución del servidor determina si sirve la predicción utilizando un motor bajo demanda (solicitud / respuesta) o el motor por lotes. En el motor por lotes, un planificador se encarga de asignar recursos para manejar ejecuciones concurrentes. Un *front-end* devuelve el resultado (predicción) al cliente una vez que se completan todas las etapas.

Miguel et al. proponen Marvin [36], una plataforma de código abierto para entrenar e

implementar modelos predictivos. Con Marvin, los modelos predictivos son construidos y luego cargados por un servidor de predicción (proporcionado por Marvin). El servidor de predicción hace que un modelo predictivo sea accesible mediante llamadas HTTP REST. De esta manera, las aplicaciones en producción pueden solicitar predicciones. Marvin también puede explotar herramientas de procesamiento de datos paralelos ejecutándose sobre herramientas comunes, como Spark. Las aplicaciones de Marvin se denominan motores, compuestas de código fuente y un archivo con parámetros y metadatos de la aplicación. Actualmente, los motores Marvin se pueden desarrollar utilizando el lenguaje Python.

Wang et al. presentan Rafiki [37], un sistema distribuido para entrenar y desplegar modelos predictivos. El desarrollo de modelos y su implementación con Rafiki se puede hacer con API RESTful o un SDK de Python. Para construir un modelo, los datos deben cargarse en el sistema de almacenamiento distribuido (HDFS) utilizado por Rafiki. Luego, se debe crear una tarea de entrenamiento del modelo donde el usuario indique el nombre de la aplicación, la ruta de datos y las formas de datos de entrada / salida. Una vez finalizada la tarea de entrenamiento, se selecciona el modelo y se puede implementar a través de un servidor HTTP usando Flask [38] con una API RESTful. Flask, es un marco de aplicación web escrito en Python para construir servicios web utilizando la interfaz de puerta de enlace de servidor web (WSGI).

Motores de *scoring* (*scoring engines*). En este enfoque, se utiliza un software especializado conocido como motor de *scoring*. El motor de *scoring* utiliza modelos descritos en un formato estándar. Una ventaja de este enfoque es que el proceso de despliegue ya no requiere a la herramienta de modelado donde se creó el modelo. Los modelos predictivos pueden construirse utilizando diferentes herramientas de modelado y exportarse a un formato estándar. Estos modelos pueden ser consumidos posteriormente por un motor de *scoring* compatible con el formato utilizado anteriormente. Los motores de *scoring* cargan modelos en la memoria y utilizan un protocolo de red para recibir datos y solicitudes de

puntuación del software en producción. A su vez, el motor de *scoring* (que tiene el rol de servidor en este enfoque), ejecuta el modelo con los datos entrantes y devuelve predicciones al software que envió la solicitud (el cliente). Los formatos más utilizados para describir modelos predictivos aprendidos por máquina son PMML (*Predictive Model Markup Language*) [39], PFA (*Portable Format for Analytics*)[40] y ONNX *Open Neural Network Exchange*) [41].

El estándar PMML fue introducido por primera vez por Grossman et al. [42]. Es un lenguaje basado en XML que se utiliza para definir modelos predictivos o conjuntos de modelos predictivos. La estructura de los modelos se describe mediante un esquema XML. Admite una amplia variedad de modelos predictivos, su primera versión fue lanzada en 1998, y ha evolucionado a lo largo de los años gracias al trabajo del Grupo de Minería de Datos (*Data Mining Group* o DMG). La última versión (4.4) se lanzó en 2019.

Con la aparición de formatos estándar para describir modelos predictivos, los motores de *scoring* también comenzaron a aparecer. Por ejemplo, Chaves et al. [43] presentó Augustus, un motor de *scoring* de código abierto para archivos PMML. Esta herramienta utiliza el modelo descrito por PMML para definir entradas, salidas, parámetros y transformaciones. Augustus fue desarrollado en Python, y tiene varios componentes. Los datos se organizan en un componente llamado Unitable, que tiene una estructura similar a un *data frame*. Un *data frame* es una estructura de datos donde todos los campos están organizados en columnas que pueden ser de diferentes tipos, pero todas tienen el mismo número de filas. Augustus proporciona un mecanismo para leer datos dentro y fuera de Augustus. Con este mecanismo, Augustus puede leer y escribir datos de una secuencia, un archivo o una base de datos.

Gorea propone DeVisa [44], este sistema gestiona un repositorio de modelos utilizando archivos PMML como una base de datos XML y explota sus propiedades predictivas o descriptivas. Las aplicaciones en producción que son consumidores del sistema DeVisa expresan su solicitud a través de un lenguaje de consulta PMML especializado llamado PMQL

(Lenguaje de consulta de modelo predictivo) definido en DeVisa. La consulta PMQL busca e invoca el servicio correspondiente para aplicar el modelo a los datos entrantes y devolver los resultados al consumidor. En resumen, las aplicaciones envían un mensaje utilizando el protocolo simple de acceso a objetos (SOAP) que envuelve datos y una solicitud de *scoring* a DeVisa expresada en PMQL. La consulta de *scoring* de PMQL se transmite al componente responsable de procesar y resolver la solicitud de PMQL.

Nathan y Kathalagiri presentan Pattern [45], un lenguaje específico de dominio (DSL) para traducir modelos PMML en flujos de trabajo en cascada. *Cascading* es un proyecto de código abierto desarrollado en Java para manejar flujos de trabajo de datos a gran escala. *Cascading* es un *middleware*, una capa de abstracción para llamadas a Hadoop, HBase, Cassandra y otros *framework* de datos distribuidos. Los flujos de trabajo se pueden implementar en plataformas de computación en la nube como Amazon AWS ElasticMapReduce y Microsoft HDInsight. El modelo PMML se analiza y los componentes API se generan para implementar el flujo de trabajo requerido. Finalmente, Pattern genera un trabajo Apache Hadoop para ejecutar el modelo predictivo en paralelo.

Existen varias herramientas de análisis de datos que pueden exportar modelos a PMML. También hay varios motores de *scoring* que consumen PMML para ser utilizados como herramienta de despliegue. Un motor de *scoring* de código abierto muy popular es JPMML [46], que está construido en Java. JPMML tiene un módulo llamado JPMML Evaluator que lee un archivo PMML. Puede verificar la validez del archivo, obtener información del modelo y calificar el modelo. Una lista completa de herramientas de software (de código abierto y patentadas) para exportar modelos (como productor de PMML) o modelos de puntaje (como consumidores de PMML) se encuentra en el sitio web del Grupo de Minería de Datos (DMG) [1].

Junto con los motores de *scoring* basados en PMML, han surgido herramientas que actúan como intermediarios entre las aplicaciones y los motores de *scoring*. Por ejemplo,

[1] http://dmg.org/pmml/products.html

Tendick y Mockus proponen un diseño de software con una capa intermedia que maneja las llamadas de las aplicaciones del usuario final [47]. El software envía datos al motor de puntuación correspondiente y devuelve predicciones a la aplicación solicitante. Un enfoque similar es presentado por Heit et al. [48], donde se presenta el Modelo de implementación y marco de ejecución (MDEF). MDEF permite capturar y encapsular transformaciones de datos al agrupar los pasos y los modelos de transformación de datos descritos en PMML. La ejecución del modelo se basa en otros motores de puntuación compatibles con PMML (por ejemplo, Knime). MDEF maneja las llamadas a través de los servicios web REST o SOAP.

El Portable Format Analytics (PFA) introducido por Pivarski et al. [49] es un formato de intercambio de modelos. Básicamente es un lenguaje de programación limitado basado en JSON que usa Avro [50] para la serialización de datos. Junto con la introducción de PFA, también se presentaron dos motores de *scoring* basados en PFA. El primer motor de *scoring* se implementó en Python y se llamó Titus. El objetivo era construir modelos predictivos con las bibliotecas de aprendizaje automático de Python y exportarlos al formato PFA. El segundo motor de *scoring* se implementó en Scala y se llama Hadrian. En este enfoque, los documentos PFA se compilan en código de bytes de Java para integrarse en diferentes aplicaciones compatibles con la máquina virtual Java.

ONNX es otro estándar para representar modelos predictivos construidos utilizando técnicas de *machine learning*, que proporciona una definición de cálculos como un modelo gráfico. El grafo está compuesto por una lista de nodos donde cada nodo tiene una o más entradas y una o más salidas. Cada nodo llama a un operador. Existe un conjunto de operadores integrados y puede ampliarse para operadores disponibles no nativos. ONNX está diseñado para admitir la descripción de modelos construidos con diferentes técnicas de aprendizaje automático. Sin embargo, se usa principalmente para modelos de redes neuronales profundas. Microsoft creó un motor de puntuación para los modelos ONNX llamado ONNX Runtime, que es compatible con los lenguajes Python y C/C# para ser utilizados por

el CPU o la GPU. Los modelos predictivos se pueden exportar al formato ONNX utilizando una variedad de herramientas, como Keras, Tensorflow, Scikit-Learn y Apple Core ML. Los modelos ONNX pueden cargarse y ejecutarse utilizando llamadas API, actualmente ONNX Runtime tiene APIs en Python, C#, C y C++ [51].

Soluciones en la nube *cloud-based*. Este tipo de enfoque cliente-servidor aprovecha la infraestructura o las arquitecturas en la nube para implementar modelos predictivos. Identificamos dos variaciones principales dentro de la implementación en la nube. La primera variación comprende herramientas de modelado o extensiones de ellas para implementar modelos predictivos como servicios web. La segunda variación son las plataformas basadas en la nube que usan infraestructura propietaria para construir e implementar modelos predictivos.

Microsoft desarrolló AzureML [52], una plataforma que proporciona un entorno de construcción de modelos y un entorno de publicación de modelos. Dentro del entorno AzureML se pueden usar módulos de aprendizaje automático de código abierto y propietarios. AzureML tiene una funcionalidad llamada *push button operationalization*, que se usa para implementar modelos predictivos como servicios web. Se pueden generar predicciones para entradas individuales o por lotes. Los módulos desarrollados en lenguaje R y Python también son compatibles con AzureML.

Bhattacharjee et al. creó Stratum [53], un *framework* para construir, implementar y administrar modelos predictivos en entornos *cloud* y *edge*. Stratum contiene un marco de desarrollo de modelo predictivo para los desarrolladores. La herramienta Stratum permite utilizar una variedad de fuentes de datos y encapsula las herramientas de aprendizaje automático en contenedores a través de una herramienta de abstracción. Una vez que el modelo predictivo está listo para implementarse, los componentes de la aplicación donde se creó el modelo se pueden exponer como una API RESTful. Stratum admite la implementación en la nube *cloud*, la niebla*fog* y los entornos de *edge computing*. Stratum tiene

un módulo para transferir modelos entrenados a máquinas destino mediante la generación de código fuente mediante el uso de plantillas específicas de la aplicación y una base de conocimiento.

Contenedores. Un contenedor es una pieza de software que encapsula el código, la configuración y todas las dependencias de software en bloques de construcción. Un contenedor es una máquina virtual ligera que se puede guardar en una máquina para utilizarla de manera transparente en otra máquina. Herramientas como Docker han facilitado el uso de contenedores al permitir construir, compartir y ejecutar contenedores en diferentes máquinas. Dado que un contenedor permite replicar fácilmente entornos, una aplicación puede ejecutarse de manera confiable en diferentes entornos informáticos. En este enfoque, los modelos predictivos se desarrollan en diferentes herramientas de modelado, con una configuración y dependencias específicas (módulos o paquetes). Una vez que el modelo está listo, todas las bibliotecas, paquetes o dependencias necesarias junto con el código desarrollado se empaquetan en un contenedor. El contenedor se configura como un servidor y las aplicaciones de usuario final envían datos y solicitudes de predicción a través de RPC o servicios web.

A diferencia de las herramientas generales para construir e implementar modelos predictivos, Clipper [54] se desarrolló exclusivamente para realizar el despliegue de modelos predictivos. Clipper se describe como un sistema de servicio de predicción. Es un intermediario entre las aplicaciones de usuario final y las herramientas de aprendizaje automático. Clipper tiene una arquitectura modular con el objetivo de implementar modelos utilizando una variedad de herramientas y aplicaciones. Clipper se divide en dos capas; la capa de abstracción del modelo y la capa de selección del modelo. La capa de abstracción del modelo expone una API común para abstraer la comunicación entre las herramientas y modelos predictivos. La capa de selección de modelo selecciona o combina dinámicamente predicciones entre modelos competidores. Las aplicaciones envían solicitudes de predicción a

Clipper mediante llamadas a la API REST. Las solicitudes son procesadas por la capa de selección de modelo. Según la solicitud, la capa de selección de modelo envía la solicitud de predicción a uno o más de los modelos a través de la capa de abstracción del modelo. Si el modelo está en caché, se generan las predicciones; de lo contrario, se utiliza un RPC para enviar la solicitud de predicción al lugar donde se aloja el modelo. Cada modelo está alojado en un contenedor Docker separado que genera las predicciones. Las predicciones se envían a través de cada capa a la aplicación del usuario final.

Zhao et al. presentan Zoo System [55] [56], que proporciona un lenguaje específico de dominio (DSL) para permitir una construcción fácil de diferentes servicios de análisis de datos. Zoo tiene tres métodos principales de implementación; El primer enfoque de implementación es utilizando contenedores Docker, cada contenedor proporciona una API RESTful para que las aplicaciones soliciten predicciones. En la segunda opción, los modelos se exportan a código JavaScript. Esto permite realizar análisis de datos complejos en un navegador web. El tercer enfoque es MirageOS [57], que construye pequeñas máquinas virtuales con un sistema operativo mínimo que aloja solo una aplicación.

Una herramienta similar a Clipper, de una manera que sirve como intermediario entre las aplicaciones de usuario final y las herramientas de aprendizaje automático, es Acumos [58]. Acumos empaqueta modelos predictivos en microservicios en contenedores portátiles que se pueden compartir a través de un catálogo. En Acumos, los usuarios cargan modelos predictivos. Luego, los modelos se publican en la plataforma Acumos donde los usuarios agregan metadatos como la descripción de la función, los formatos de entrada / salida y la categoría del modelo. La plataforma Acumos incluye modelos cargados como microservicios en una imagen Docker lista para ser implementada. Los consumidores descargan e implementan el servicio en un contenedor Docker. Una vez implementado, los usuarios envían datos como entrada al microservicio y reciben su salida a través de una API RESTful.

MLflow es una herramienta de código abierto para desarrollar e implementar modelos

predictivos [59]. MLflow utiliza contenedores y tiene un componente que permite empaquetar modelos predictivos en múltiples formatos. Por ejemplo, los modelos se pueden implementar como contenedores de Docker para llamarlos a través de la API REST, como una función definida por el usuario de Apache Spark, o en plataformas de servicio administradas en la nube como Amazon SageMaker o Azure ML.

Data and Learning Hub for Science (DLHub) [60] es una plataforma para publicar, compartir, descubrir y reutilizar modelos predictivos. Una de sus principales funcionalidades es la implementación del modelo. DLHub admite el lenguaje Python y una interfaz de línea de comandos. Los modelos se definen mediante un esquema que captura su descripción, entorno computacional e interfaz. El entorno computacional describe las dependencias de software y los archivos requeridos. La interfaz describe el formato de las entradas / salidas del modelo. Los modelos publicados en DLHub se pueden convertir en un contenedor ejecutable que implementa una interfaz estándar. Estos contenedores encapsulan todas las dependencias requeridas para llamar al modelo para predicciones a pedido.

2.2.2 Enfoque de despliegue en sistemas gestores de bases de datos

En este enfoque, los modelos predictivos se implementan dentro de un Sistema gestor de base de datos (en inglés Data Base Management System o DBMS). Al tener la ejecución de los modelos dentro de un DBMS, todas las propiedades de gestión de datos y arquitecturas paralelas que ofrecen los DBMS se pueden usar [61]. Hay algunos sistemas de bases de datos que cuentan con algunas funcionalidades analíticas. Esto significa que los modelos pueden ser entrenados y desplegados dentro del DBMS directamente sin usar herramientas externas. Por ejemplo; Base de datos como SQL de Azure, IBM BigR, SparkR, Oracle Data Mining u Oracle R Enterprise. Sin embargo, este enfoque tiene limitaciones; Una de las más importantes es que los científicos de datos suelen utilizar un conjunto de herramientas especializadas para la transformación de datos y el modelado predictivo. Por lo tanto, están restringidos por las técnicas de aprendizaje automático disponibles proporcionadas por el

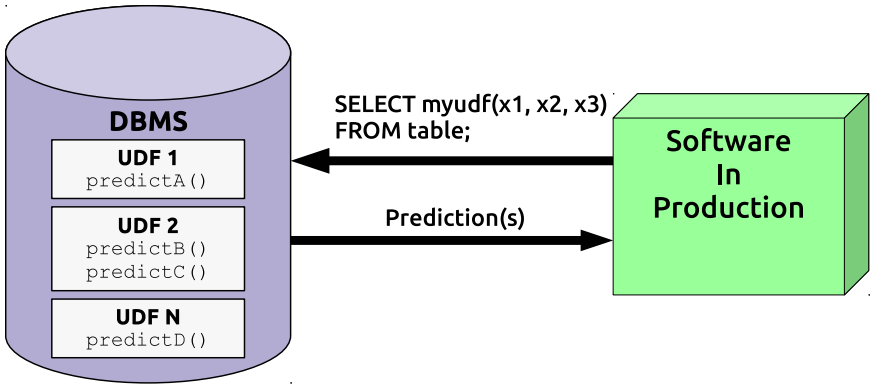


Figura 2.2: Vista gráfica de los modelos predictivos implementados dentro de una base de datos utilizando una función definida por el usuario (UDF). El software en producción invoca a una consulta utilizando SQL. Posteriormente, la base de datos ejecuta la UDF que contiene un modelo predictivo y devuelve las predicciones como una tabla de registros.

proveedor de DBMS. Otro enfoque es construir modelos predictivos fuera del DBMS e importarlos más tarde mediante el uso de un script SQL o una función definida por el usuario (en inglés User Defined Function o UDF).

Cuando se utiliza el enfoque de despliegue en bases de datos, las predicciones se calculan dentro del DBMS mediante un procedimiento almacenado o un UDF que recibe los datos como una tabla y devuelve las predicciones también como una tabla. Una ventaja de este enfoque con respecto al uso de herramientas analíticas es que los datos ya están dentro de la base de datos, por lo que los cálculos del modelo predictivo se realizan cerca de donde residen los datos. Esto significa que no hay un tiempo extra para extraer datos de la base de datos para procesar / modelar en otra herramienta y almacenar los resultados en la base de datos. La figura 2.2 muestra un esquema del enfoque de despliegue en la base de datos para implementar modelos predictivos.

Ha habido muchas propuestas en el pasado extender las funcionalidades de los DBMS y el lenguaje SQL con capacidades de modelado de aprendizaje automático. Algunas de las propuestas incluso se remontan a tiempos en que las tareas de modelado predictivo se conocían comúnmente como minería de datos o descubrimiento de conocimiento, como en [62, 63, 64, 65, 66, 67, 68]. Algunas propuestas extendieron el SQL con nuevos operado-

res o comandos. Los scripts de SQL se pueden usar para expresar modelos relativamente simples, mientras que para modelos más complejos se utilizan funciones definidas por el usuario. Ahora describimos algunos de los enfoques más relevantes centrados en los aspectos técnicos de la implementación del modelo predictivo dentro de un DBMS.

Das et al. presentan una propuesta que combina el uso de estándares para la descripción del modelo, como PMML, y la implementación en la base de datos. [69]. La base de datos EMC Greenplum se usa junto con una herramienta comercial llamada Zementis Universal PMML plug-in. Este complemento es un motor de *scoring* donde los modelos predictivos descritos en un archivo PMML se asignan a una función SQL. La definición de la función SQL coincide con el modelo predictivo correspondiente. Este enfoque funciona de la siguiente manera. Primero, el archivo PMML se copia en la instalación de Greenplum y se crea la función SQL. Luego, cuando se invoca la función SQL correspondiente, el modelo se carga y ejecuta para obtener predicciones dentro de la base de datos. De esta manera, los modelos se pueden crear con cualquier herramienta de modelado compatible con PMML y desplegarse dentro de una base de datos.

Una alternativa de código abierto para la implementación en la base de datos es MADlib [70]. Esta herramienta proporciona un conjunto de algoritmos basados en SQL para aprendizaje automático, minería de datos y estadísticas dentro de una base de datos. MADlib es compatible con las bases de datos PostgreSQL y Greenplum, y admite muchos algoritmos de aprendizaje supervisados, tales como regresión lineal, regresión logística, naive Bayes, árboles de decisión y máquinas de vectores de soporte. MADlib también admite algoritmos de aprendizaje no supervisados. MADlib contiene scripts SQL y / o funciones definidas por el usuario para entrenar e implementar modelos. Una vez que se crean los modelos, se conservan dentro de la base de datos y se pueden invocar mediante una consulta SQL. Los modelos creados con MADlib se pueden exportar al formato PMML, aunque los modelos PMML no se pueden importar a MADlib.

Una solución que pretende combinar consultas relacionales con un motor similar a

MapReduce que implementa técnicas de aprendizaje automático es Shark [71]. Shark permite definir algoritmos de aprendizaje automático a través del lenguaje SQL en un entorno Mapreduce. Actualmente, Shark utiliza Spark como motor de ejecución y ha implementado algoritmos básicos de aprendizaje automático para construir modelos predictivos. Una vez que se crean los modelos, se pueden implementar en un clúster de Spark para invocarlos mediante una consulta SQL.

La integración de herramientas de modelado con un DBMS también es un enfoque comúnmente utilizado. Prasad et al. presentan una versión especial de R integrada con la base de datos HP Vertica, llamado Distributed R [72]. En esta herramienta, Vertica y R se comunican a través de funciones definidas por el usuario y escritas usando el lenguaje R. Los datos seleccionados en una consulta SQL en Vertica se pasan a R como un objeto de marco de datos. En R, el marco de datos se puede manipular y usar como entrada para muchos algoritmos de aprendizaje automático para construir modelos predictivos. Este enfoque elimina la necesidad de conectar Vertica con R a través de ODBC. Distributed R permite la ejecución de algoritmos paralelos de aprendizaje automático y extiende las funcionalidades de R con todas las ventajas de la gestión de datos y la escalabilidad que ofrece un DBMS. Los modelos construidos en Distributed R pueden implementarse en todos los nodos Vertica disponibles y pueden invocarse como una función definida por el usuario.

Mahajan et al. proponen una herramienta para unir bases de datos, aprendizaje automático y diseño de hardware. Proponen aceleración en la base de datos de análisis avanzado (DAnA) [73], que ofrece dos características principales; 1) una interfaz de programación para el modelado predictivo y 2) la aceleración de la ejecución de modelos predictivos en una base de datos mediante el uso de una matriz de una FPGA (en inglés Field-Programmable Gate Array). Al usar DAnA, los algoritmos de *machine learning* se pueden expresar usando una extensión del lenguaje Python con un lenguaje específico de dominio (DSL) incrustado en Python. El código de Python se declara como una UDF para ser llamado por consultas SQL. DAnA traduce el UDF de Python para que se implemente en un

motor de *scoring* utilizando un FPGA conectado al DBMS para ejecutarlo.

Schüle et al. proponen MLearn y ML2SQL para construir y desplegar modelos predictivos en bases de datos [74]. MLearn es un lenguaje y una herramienta de modelado para realizar la manipulación de datos, el entrenamiento de modelos y la validación de modelos. MLearn proporciona desarrollo bloques de aprendizaje de la máquina. La herramienta ML2SQL es un compilador que traduce los modelos construidos con MLearn a Python o SQL para ser desplegado en las bases de datos PostgreSQL o Hyper. Por lo tanto, los modelos predictivos se importan a las bases de datos como funciones definidas por el usuario y llama a través de consultas SQL.

2.2.3 Otros

Existe una amplia variedad de entornos de implementación para predicción modelos. Por lo tanto, los enfoques de implementación adoptados pueden ser muy diferentes. Aunque hemos seleccionado y agrupado trabajos que proponen los enfoques de despliegue cliente-servidor y en sistemas gestores de base de datos, todavía hay propuestas que no corresponden en estas categorías. Por lo tanto, los agrupamos en esta subsección.

Las empresas del ramo tecnológico a menudo optan por construir sus propias herramientas para realizar tareas relacionadas con el aprendizaje automático. Por ejemplo, Facebook tiene una gran variedad modelos predictivos construidos con diferentes herramientas de aprendizaje automático que deben implementarse en varios entornos. En un intento por simplificar el desarrollo de aplicaciones relacionadas con el aprendizaje automático, Facebook ha desarrollado FBLearner. Esta es una plataforma que permite realizar diferentes tareas de aprendizaje automático, como transformaciones de datos, ingeniería de características (*feature engineering*), construcción de modelos e implementación de modelos. Su plataforma admite diferentes algoritmos de aprendizaje automático, como regresión logística, máquinas de vectores de soporte, árboles de decisión potenciados por gradiente y redes neuronales profundas. Caffe2 y Pytorch también son herramientas de uso común dentro de

los científicos de datos de Facebook. FBLearner se compone de tres módulos; FBLearner Feature Store, donde se recopilan y transforman los datos, FBLearner Flow, su propia herramienta para construir y evaluar modelos predictivos, y FBLearner predictor, que es un motor de inferencia que utiliza modelos construidos dentro de FBLearner para proporcionar predicciones a las aplicaciones finales. FBLearner también ofrece compatibilidad con el formato Open Neural Network Exchange (ONNX) como herramienta de entorno de producción para implementar modelos en arquitecturas de computadora utilizando CPU, GPU u otro hardware especializado [75, 76, 77].

Uber desarrolló Michelangelo, una plataforma interna como servicio para construir, implementar y usar modelos predictivos aprendidos por máquina. La plataforma Michelangelo se compone de muchas herramientas de código abierto e internas para construir modelos predictivos. Actualmente, Michelangelo ofrece tres opciones de implementación diferentes. La primera opción es *offline*, donde los modelos se implementan en un contenedor para ejecutarse con Spark a pedido o en un horario. La segunda opción es en línea, donde los modelos se implementan en un servicio de predicción en línea que se puede llamar a través de llamadas RPC. La tercera y última opción es la implementación del modelo en una biblioteca. En donde un modelo se integra como una biblioteca en otro servicio y se puede invocar a través de una API Java [78, 79].

El enfoque cliente-servidor no es compatible con el despliegue de modelos predictivos en dispositivos móviles o IoT (*edge-computing*). Esto se debe a que muchos dispositivos de borde tienen una red limitada o nula y los modelos predictivos pueden requerir generar predicciones en el campo. Un enfoque para resolver este problema es utilizar un compilador para traducir los modelos predictivos a un idioma de destino. El objetivo es integrar el modelo predictivo en el software del dispositivo. Por ejemplo, Barbareschi et al. [80] implementa los modelos de árbol de decisión descritos en un PMML mediante la creación de un compilador que los traduce al código VHDL. Luego, el código VHDL se puede implementar en FPGA. También Gopinath et al. [81] presenta SEEDOT, un lenguaje de dominio

específico para modelos predictivos y un compilador que genera código fuente C de punto fijo para ejecutarse en dispositivos IoT restringidos.

2.3 Compiladores

Un compilador es un programa de computadora que lee una secuencia que representa un programa de computadora escrito en un idioma (la fuente) y lo traduce a un programa equivalente a otro idioma (el destino). Como parte importante de este proceso de traducción, el compilador informa a su usuario de la presencia de errores en el programa fuente [82].

La construcción de un compilador requiere el uso de técnicas de lenguajes de programación, arquitectura de computadoras, teoría del lenguaje, algoritmos e ingeniería de software. Existe una amplia variedad de lenguajes de programación que pueden ser de origen o de destino. Todos los compiladores siguen tareas básicas que deben realizar. Estas tareas se organizan en fases, cada fase opera en un lenguaje abstracto diferente. En software, un lenguaje abstracto es un modelo que representa cómputo sin las reglas de sintaxis de un lenguaje de programación concreto.

Muchos compiladores modernos tienen un diseño de dos etapas que consiste en un *front-end* y un *back-end*. El *front-end* traduce el lenguaje fuente a una representación intermedia o una serie de representaciones intermedias. La representación intermedia es en gran medida independiente de de la máquina o lenguaje destino. El *back-end* recibe la representación intermedia generada por el *front-end*. El *back-end* incluye todas las piezas del compilador que dependen del idioma de destino. Este enfoque ayuda a separar las preocupaciones entre las representaciones del idioma de origen y de destino. Con este tipo de diseño, el *front-end* se puede reutilizar con un *back-end* diferente para producir un compilador para el mismo idioma de origen para un idioma de destino diferente.

Si el *back-end* se diseña con cuidado, agregar un nuevo idioma de destino diferente requeriría modificaciones mínimas en el *back-end*. Lo mismo podría decirse sobre la tra-

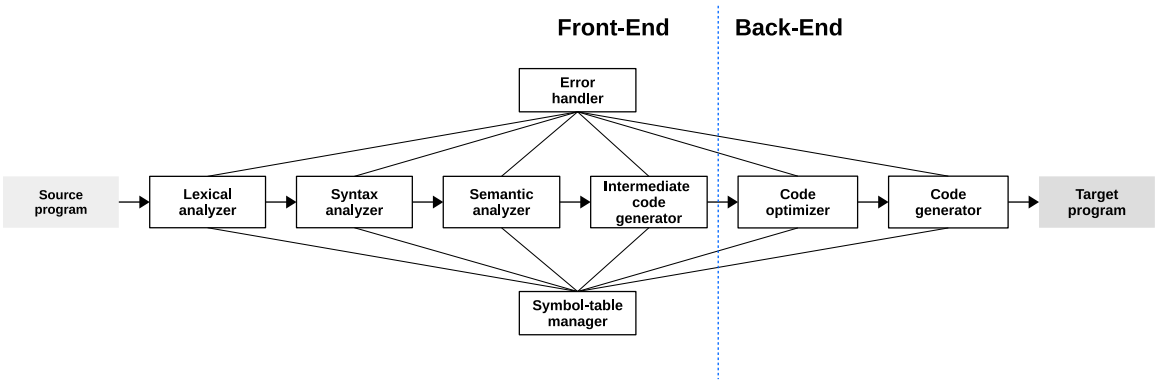


Figura 2.3: Fases típicas de un compilador.

ducción de diferentes idiomas de origen en la misma representación intermedia. Se puede utilizar el mismo *back-end*, obteniendo diferentes compiladores para el mismo idioma de destino. Sin embargo, este enfoque es limitado debido a diferencias sutiles en los puntos de vista de diferentes idiomas [82]. Por lo tanto, a medida que aumenta el número de idiomas de origen y destino, también aumenta la complejidad del lenguaje intermedio y la construcción del compilador. Una estructura típica de las fases del compilador se muestra en la Figura 2.3.

Una compilación generalmente se implementa como una secuencia de transformaciones [83]:

$$(SL, L_1), (L_1, L_2), ..., (L_k, TL) \tag{2.1}$$

Donde SL es el idioma de origen y TL es el idioma de destino. Cada idioma L_i se llama *lenguaje intermedio o representación intermedia*. El proceso de compilación se puede dividir en análisis y síntesis. El *front-end* del compilador realiza el análisis y el *back-end* realiza la síntesis. El análisis consiste en descubrir la estructura y determinar el significado del programa fuente. La síntesis se encarga de crear el programa de destino que es semánticamente equivalente al programa fuente. El análisis generalmente se divide en analizadores léxicos, sintácticos y semánticos. La síntesis se divide en generador de código intermedio, optimización de código y generador de código. Estas tareas se detallan en la Tabla 2.2.

Los compiladores se utilizaron inicialmente para traducir lenguajes de alto nivel (legi-

Etapa - Módulo del compilador	**Fase**	**Descripción**
Analísis - *Front-End*	Analizador léxico	En esta fase, se realizan las siguientes tareas. La entrada se divide en palabras individuales o ”tokens”. Se procesa la secuencia de caracteres de la entrada. Y se produce una secuencia de nombres, palabras clave y signos de puntuación. Además, se descartan los espacios en blanco y los comentarios entre los tokens.
	Analizador Sintáctico	También llamado análisis, esta fase implica agrupar los tokens del programa fuente en frases gramaticales que el compilador utiliza para sintetizar la salida.
	Analizador semántico	En esta fase queremos determinar lo siguiente: qué significa cada frase, definir los usos de las variables con respecto a sus definiciones, verificar los tipos de expresiones y solicitar la traducción de cada frase.
	Generador de código intermedio	En esta fase, se produce una representación intermedia del programa fuente. Esta es una representación abstracta y debe ser fácil de producir y fácil de traducir al programa de destino.
Síntesis - *Back-End*	Optimización de código	La tarea realizada en esta fase intenta mejorar el código intermedio mediante el análisis de instrucciones y la modificación del código intermedio. Por ejemplo: instrucciones de simplificación, eliminación de código muerto u optimizaciones de bucle.
	Generador de código	La fase final en el compilador, toma como entrada una representación intermedia del programa fuente y produce como salida un programa objetivo equivalente.

Tabla 2.2: Descripción de las fases típicas de una compilación.

bles para humanos) a bajos lenguaje de nivel (legible por computadoras). Hoy en día hay muchos tipos diferentes de compiladores, como los siguientes.

- Compilador cruzado: este tipo de compilador traduce el programa fuente para que se ejecute en una arquitectura de máquina o sistema operativo diferente del que se ejecuta el compilador.
- Compilador Bootstrap: también conocido como bootstrapping, se refiere a un compilador escrito en el lenguaje fuente que tiene la intención de compilar.

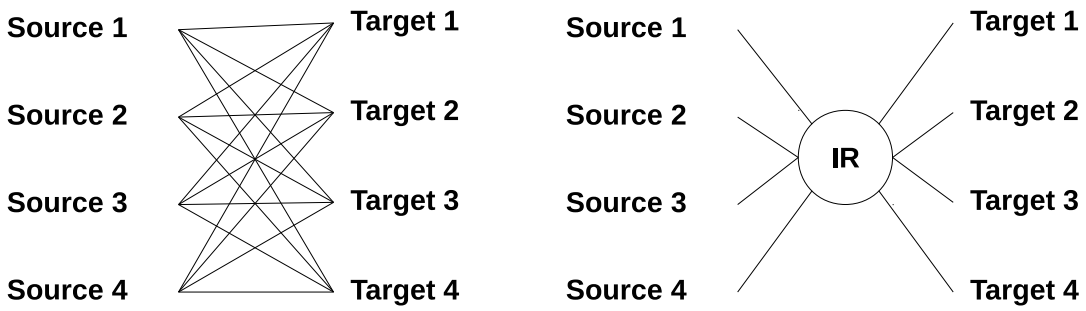


Figura 2.4: Casos de traducción de diferentes idiomas de origen y destino con y sin una representación intermedia. Cada línea representa una traducción.

- Decompiler: este tipo de compilador recibe un archivo ejecutable como entrada e intenta generar las instrucciones en un lenguaje de alto nivel para poder recompilarlo.

- Source to source: se refiere a los compiladores donde el programa fuente se traduce a otro lenguaje de computadora que tiene aproximadamente el mismo nivel de abstracción [84].

2.3.1 Representación intermedia

Una representación intermedia (IR) es un tipo de lenguaje abstracto que expresa las operaciones del lenguaje fuente sin comprometer demasiado los detalles específicos del lenguaje destino. También, es independiente de los detalles del lenguaje fuente. Los compiladores usan diferentes tipos de IR dependiendo de las necesidades del compilador, como el lenguaje de origen, lenguaje destino y las transformaciones que aplica el compilador.

Si queremos compilar N lenguajes fuente a M lenguajes destino directamente (sin un IR) necesitaremos construir compiladores de $N \times M$. Un compilador portable traduce el lenguaje fuente a un IR y luego traduce el IR al lenguaje destino. Como se ilustra en la Figura 2.4, con una IR N *front-ends* y M *back-ends* son necesarios para cumplir la misma tarea [85].

Hay muchos tipos de representaciones intermedias. Cada IR tiene sus propias ventajas. Una diferencia principal entre los tipos de IR es su nivel de abstracción del lenguaje. Una

representación de alto nivel está muy cerca del lenguaje fuente; Se produce fácilmente a partir del código fuente y se manipula en el *front-end* para modificaciones u optimizaciones. Una representación de bajo nivel está muy cerca del lenguaje destino, lo que la hace adecuada para la manipulación u optimización de acuerdo con el lenguaje destino.

Seleccionar una representación intermedia para un compilador que sea apropiada requiere una comprensión adecuada de los lenguajes origen y destino. Los compiladores *source-to-source* pueden usar una representación intermedia que se parezca mucho al código fuente. Un compilador que produce código ensamblador para un microcontrolador podría obtener mejores resultados con una representación intermedia similar a un código ensamblador. En términos generales, las representaciones intermedias se dividen en las tres categorías siguientes [86].

- Representación intermedia basada en grafos: codifican el conocimiento del compilador en un grafo. Las IR basados en grafos representan programas con objetos gráficos como nodos, bordes, listas o árboles.

- Representación intermedia lineal: se asemejan a pseudocódigo para alguna máquina abstracta. Los algoritmos iteran sobre secuencias simples y lineales de operaciones. Consiste en una secuencia de instrucciones que se ejecutan en su orden de aparición.

- Representación intermedia híbrida: esta categoría combina elementos de IRs tanto de grafos como lineales, en un intento de capturar sus fortalezas y evitar sus debilidades.

2.4 Oportunidades de Investigación

Los enfoques de despliegue basados en cliente-servidor dependen de la disponibilidad de una red. Usar una herramienta de modelado para realizar la tarea de despliegue puede ser una opción rápida y fácil. Sin embargo, las herramientas de modelado no son adecuadas para cumplir requisitos de alto rendimiento u ofrecer escalabilidad. En el enfoque

cliente-servidor, el servidor requiere replicar el entorno donde se construyeron los modelos. Para replicar este entorno, se deben instalar versiones compatibles de las herramientas de modelado, paquetes y dependencias en el servidor. A menudo, se requieren bibliotecas o paquetes adicionales para exponer los modelos predictivos a un protocolo de comunicación para ser utilizado por el software en la producción. Las tareas de despliegue y mantenimiento se vuelven más difíciles de administrar cuando muchos modelos desarrollados en diferentes herramientas de modelado se implementan constantemente. La replicabilidad de la configuración del servidor es una tarea difícil que puede mitigarse mediante el uso de contenedores. Los contenedores encapsulan el software, los paquetes y las dependencias reales para realizar los cálculos requeridos por los modelos predictivos.

Los motores de *scoring* permiten el despliegue de modelos descritos en algún formato estándar. Este enfoque desacopla el modelado del despliegue. Esto significa que los científicos de datos pueden usar diferentes herramientas de análisis de datos para construir modelos predictivos y exportarlos a un formato específico para su posterior implementación. Sin embargo, este enfoque también se basa en la disponibilidad de una red. Las soluciones en la nube también ofrecen una opción fácil y rápida para la implementación, pero el conjunto de algoritmos de entrenamiento de modelos disponibles es limitado y depende del proveedor de los servicios en la nube. Otra desventaja de usar soluciones en la nube es que los modelos deben construirse dentro de una plataforma en la nube. Esto significa que todos los datos deben cargarse en la nube para la capacitación del modelo, lo que no siempre es factible debido a problemas de seguridad o limitaciones de la red.

El enfoque de despliegue de modelos predictivos en un sistema gestor de base de datos ofrece grandes ventajas relacionadas con la gestión de datos. Además, la sobrecarga de llevar y sacar datos de la base de datos no está presente en este enfoque. Algunos sistemas de gestores de bases de datos incluyen su propio conjunto de herramientas de modelado. Desafortunadamente, esto limita a los científicos de datos. Normalmente, los científicos de datos usan bases de datos solo para recopilar datos y / o almacenar resultados, y optan por

usar herramientas especializadas como R, Python, SPSS o SAS. Los sistemas gestores de bases de datos, además de las licencias, a veces requieren hardware especial para funcionar correctamente, lo que debe tenerse en cuenta al diseñar la infraestructura del servidor de despliegue.

La mayoría de las herramientas propuestas revisadas en esta sección son soluciones integrales para el modelado predictivo. Esto significa que ofrecen funcionalidades generales para procesar datos, construir, validar e implementar modelos predictivos. Dado que son herramientas generales, a menudo la opción disponible es conectar a través de algún mecanismo de comunicación estas herramientas con el software en producción. Esto deja un vacío en términos de disponibilidad de herramientas especializadas para la implementación. Dado que existe un amplio espectro de posibles entornos de implementación, es difícil encontrar una herramienta que automatice esta tarea y, al mismo tiempo, cumpla con los requisitos de software específicos con respecto a la eficiencia de tiempo y espacio del software operativo.

Hemos identificado la falta de herramientas especializadas para el despliegue de modelos predictivos. Especialmente, identificamos la falta de herramientas para implementar modelos predictivos que cumplan con los siguientes requisitos; a) el módulo de predicción modelo es autónomo y no requiere una red u otras dependencias, b) el módulo de predicción modelo puede integrarse como parte de un sistema más grande en producción utilizando el mismo lenguaje de programación, y c) las predicciones se calculan de manera eficiente de acuerdo con la máquina de destino. Una forma de cumplir con los requisitos antes mencionados es tener el código fuente necesario para implementar modelos. Aunque se pueden codificar manualmente los modelos predictivos en un lenguaje de programación de computadora para su implementación, es una tarea laboriosa y propensa a errores. Además, la codificación manual de modelos no es factible a escala industrial, ya que se requiere que muchos modelos estén en producción y se actualicen constantemente. Sin embargo, dado que los modelos predictivos pueden definirse formalmente (y son generados por compu-

tadora), podemos transformarlos algorítmicamente utilizando herramientas informáticas.

En el siguiente capítulo, presentamos nuestra propuesta; un compilador multi-destino para generar código a partir de una descripción de un modelo predictivo. Los compiladores realizan tareas repetitivas y formalmente definidas que, de lo contrario, de realizarse manualmente requerirían mucho esfuerzo y serían propensos a errores. Además, el diseño del compilador propuesto es multi-destino. Esto permite la generación de código fuente para diferentes idiomas de destino. Al tener esta característica de multi-destino en el compilador, podemos implementar modelos predictivos en diferentes entornos de producción. Además, podemos generar código para implementar en diferentes arquitecturas de computadora para que las predicciones se computen de manera eficiente de acuerdo con el hardware subyacente. En el Capítulo 4 presentamos una implementación del compilador propuesto. Además, el Capítulo 5 presenta los resultados experimentales que prueban la viabilidad de nuestra propuesta y muestran el rendimiento del código generado.

CAPÍTULO 3
COMPILADOR MULTI-DESTINO PARA EL DESPLIEGUE DE MODELOS PREDICTIVOS

En este capítulo, se presenta el compilador multi-destino para el despliegue de modelos predictivos. Este compilador tiene un diseño modular para generar código para diferentes lenguajes de programación. La estructura general y el diseño del compilador se presentan en la Sección 3.1. El *front-end*; donde las descripciones formales de los modelos predictivos se traducen en la representación intermedia se presenta en la Sección 3.2. La representación intermedia se presenta en la Sección 3.3. Finalmente, en la Sección 3.4 se presenta el *back-end* del compilador, donde se genera el código fuente.

3.1 Diseño general del compilador multi-destino

Cada herramienta de modelado tiene su propia representación interna definida para un modelo predictivo dado. Además, después de construir los modelos, la mayoría de las herramientas de modelado permiten almacenarlos en archivos con un formato específico. Por lo tanto, se obtiene una descripción formal generada por computadora de un modelo predictivo a partir de herramientas de modelado. Esta descripción formal se utiliza como entrada para el compilador multi-destino.

El compilador multi-destino recibe dos parámetros; un modelo predictivo y un lenguaje destino. El compilador realiza una serie de transformaciones de lenguaje y genera el código fuente que implementa el modelo predictivo dado en el lenguaje de programación especificado. La figura 3.1 muestra un esquema del compilador propuesto. En este esquema, las entradas del compilador son la descripción del modelo predictivo y el lenguaje destino (por ejemplo, un modelo lineal generalizado y CUDA-C). El compilador genera el código fuente de un programa que implementa el modelo en el lenguaje de programación especificado.

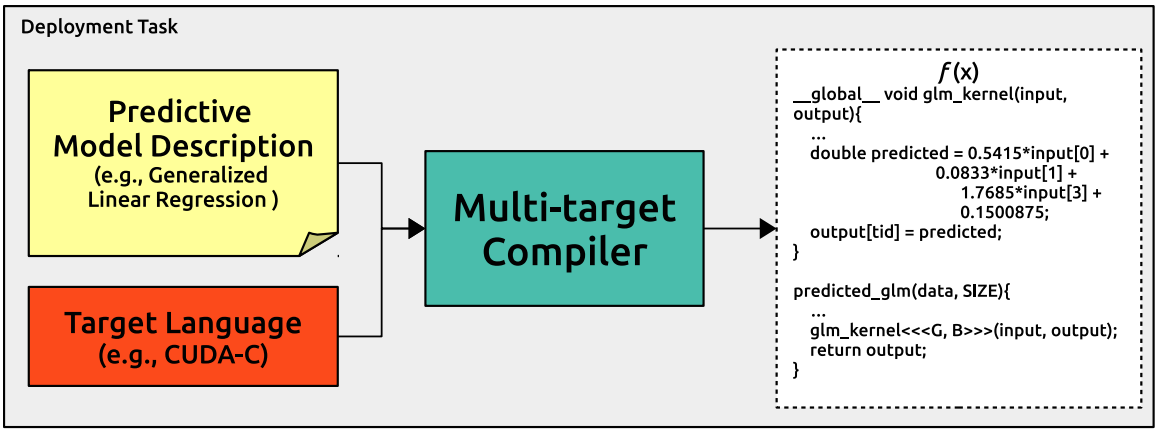


Figura 3.1: Descripción general del compilador multi-destino propuesto para el despliegue de modelos predictivos.

Debido a la gran variedad de entornos de producción en los que se podrían implementar modelos predictivos, el compilador propuesto está diseñado para ser multi-destino. Aho et al. se refirieron a esta característica de los compiladores como redireccionamiento (*retargetability*), al tener varios *back-end* que permiten traducir de un lenguaje fuente a varios lenguaje destino [82]. El atributo multi-destino del compilador propuesto permite expandir el conjunto de lenguajes de programación que se generarán. Por lo tanto, el compilador propuesto está diseñado para ser modular y flexible para admitir muchos tipos de modelos predictivos y para poder generar código fuente en diferentes lenguajes de programación de destino sin afectar el resto de su estructura. La figura 3.2 muestra la estructura general del compilador multi-destino, cuyo diseño permite tener muchos *front-end* y muchos *back-end*.

Los componentes principales del compilador propuesto son el *front-end* y el *back-end*. El compilador propuesto tiene un diseño de dos pasos. Esto significa que el compilador recorre todo el programa dos veces, uno con el *front-end* y otro con el *back-end*. Además, hemos definido una representación intermedia de código que actúa como un puente entre los *front-end* y *back-end*. La Figura 3.3 ilustra la arquitectura del compilador con los pasos realizados en el *front-end* y el *back-end*. Un *front-end* se encarga de traducir una descripción del modelo predictivo a la representación intermedia. La representación intermedia contiene toda la información requerida de un modelo predictivo dado. Un *back-end* traduce

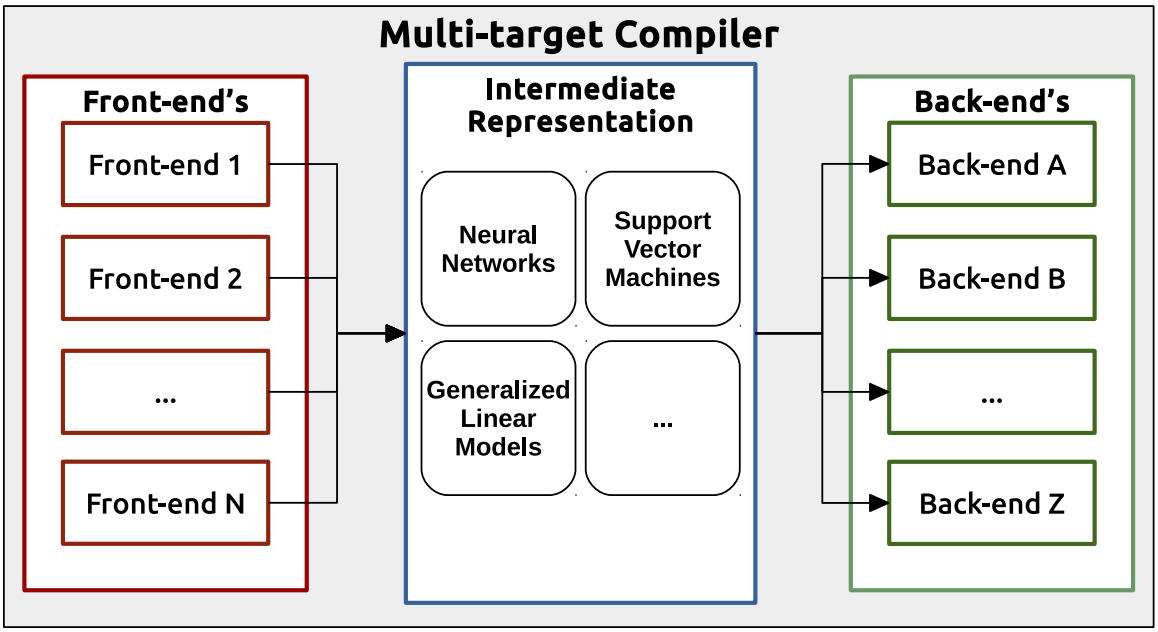


Figura 3.2: Diseño general del compilador multi-destino propuesto para el despliegue de modelos predictivos.

la representación intermedia de un modelo a un lenguaje de programación específico.

Una pieza central del compilador es la representación intermedia (IR). La IR habilita la modularidad requerida y la capacidad de agregar *back-ends* sin afectar el resto del compilador. El diseño basado en representación intermedia permite que el compilador tenga M *front-ends* y N *back-ends*, en lugar de $M \times N$ compiladores. La IR permite desarrollar un *front-end* de acuerdo con el formato de la representación de modelos predictivos, y agre-

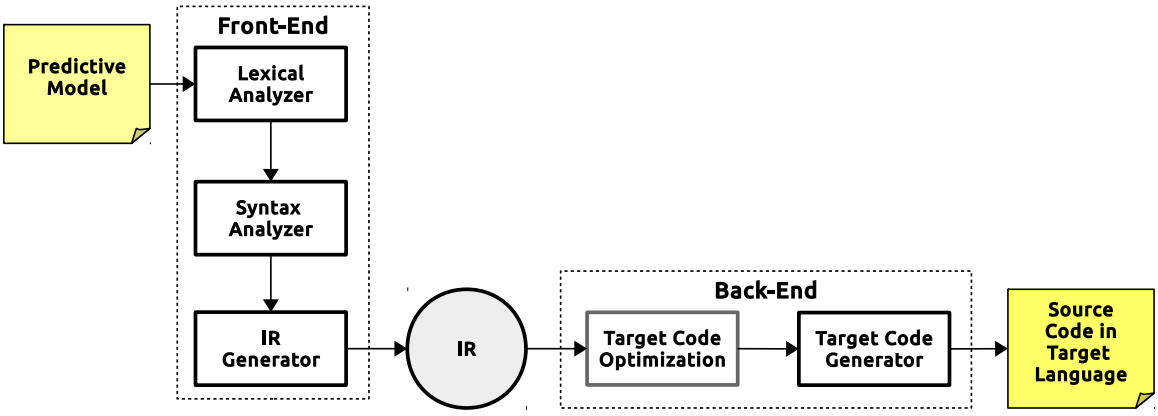


Figura 3.3: Ejemplo del compilador multi-destino propuesto que incluye las tareas realizadas en el *front-end* y en el *back-end*.

gar múltiples *back-end* según se requieran los lenguajes de programación destino. Hemos definido una gramática libre de contexto (*Context-Free Grammar* o CFG) para la representación intermedia de código. Esta CFG es un subconjunto de una gramática completa de un lenguaje de programación que se ha ampliado con expresiones específicas de *machine learning* y operaciones de álgebra lineal de uso común. La representación intermedia tiene una estructura de árbol de sintaxis abstracta (AST) donde los nodos son parte de la CGF. Esto permite resumir una estructura gramatical sin incluir detalles sobre derivaciones.

En términos generales, una descripción de un modelo predictivo contiene información sobre el modelo, sus elementos y los valores que definen un modelo. Sin embargo, no contiene un algoritmo de predicción. La descripción del modelo es declarativa. Por lo tanto, el compilador convierte la porción declarativa del modelo en una descripción procedural. Por otro lado, la estructura general de un algoritmo de predicción es siempre la misma para un tipo de modelo de aprendizaje automático específico. Por lo tanto, se pueden definir plantillas para implementar una función de predicción para cada tipo de modelo. Una plantilla contiene una estructura de elementos de la gramática que representa un programa. Dicho programa implementa el algoritmo de predicción de un modelo predictivo y tiene elementos dinámicos que se modifican con valores concretos u otros elementos de un tipo de modelo específico. Se han desarrollado plantillas para los tipos de modelos predictivos compatibles con el compilador y son parte de nuestra definición de representación intermedia. El uso de plantillas en compiladores es muy común, especialmente en los módulos de generación de código. Dado que la entrada es un modelo predictivo que representa una abstracción, las plantillas proporcionan una forma segura (sintaxis correcta) de generar el código de destino.

3.2 Front-End

Este módulo es el principal responsable de traducir una descripción de un modelo predictivo a la representación intermedia. En los compiladores *clásicos*, la entrada es una se-

cuencia de instrucciones en un lenguaje de programación dado y normalmente se realizan tres tareas de análisis: léxico, sintáctico y semántico. No obstante, debido a la naturaleza de la entrada de nuestro compilador propuesto, que es una descripción de un modelo predictivo generada por computadora, se simplifican algunas de las tareas de análisis mencionadas anteriormente.

Proponemos que cualquier *front-end* desarrollado para el compilador propuesto debe tener al menos tres módulos; un analizador léxico, un analizador de sintaxis y un generador de representación intermedia. El análisis léxico se encarga de leer la entrada y convertir una secuencia de caracteres en una secuencia de tokens. Los tokens son cadenas con un significado asignado e identificado. Posteriormente, se puede usar un módulo analizador de sintaxis para analizar los tokens y verificar que la descripción del modelo predictivo sea válida para ser procesada por el siguiente módulo. En lugar de contener un algoritmo de predicción, una descripción de modelo predictivo generalmente contiene los elementos y coeficientes que definen un modelo. Por ejemplo, un modelo de red neuronal se describe por sus capas, neuronas, enlaces entre neuronas, funciones de activación y (pesos). Por lo tanto, omitimos el análisis semántico. En cambio, después del análisis de sintaxis, proponemos usar un módulo llamado generador de IR que realiza las siguientes dos tareas. Primero, el módulo contiene la información relevante de la descripción del modelo predictivo analizado. En segundo lugar, el módulo usa las plantillas para generar una representación intermedia del código del modelo predictivo. El módulo generador de la IR utiliza una plantilla de modelo para un modelo determinado y llena los elementos dinámicos con valores concretos. Finalmente, el módulo generador de IR genera una representación intermedia de un programa que implementa la función de predicción de un modelo.

3.3 Representación intermedia

La mayoría de los algoritmos de aprendizaje automático dependen en gran medida del álgebra lineal para las tareas de entrenamiento y la predicción de modelos predictivos. Por

lo tanto, la mayoría de los algoritmos de predicción de *machine learning* pueden expresarse en términos de operaciones de álgebra lineal. A partir de la descripción matemática de un modelo, podemos definir las partes centrales de su representación intermedia. Esta representación intermedia se puede convertir en código fuente en diferentes lenguajes de programación destino específicos. En esta sección, se presenta una gramática libre de contexto para describir la estructura sintáctica de la representación intermedia de modelos predictivos. Como se señaló anteriormente, una descripción formal de los modelos predictivos no contiene el algoritmo de predicción. Por lo tanto, presentamos para cada tipo de modelo predictivo; un algoritmo de predicción y una plantilla de una representación intermedia.

3.3.1 Grámatica libre de contexto

La estructura sintáctica de la representación intermedia para el compilador multi-destino propuesto se describe mediante una gramática libre de contexto. Como se indicó anteriormente, hemos definido plantillas para cada tipo de modelo predictivo. La representación intermedia del compilador tiene una estructura de árbol de sintaxis abstracta (*abstract syntax tree* o AST) donde cada nodo del árbol denota una construcción que ocurre en el algoritmo de predicción. La gramática del compilador se presenta en la Tabla 3.1. En esta gramática, los símbolos en mayúscula no son terminales y los símbolos en minúscula son terminales. La notación N* significa que N no es terminal y puede haber 0, 1 o muchas repeticiones de N. La gramática contiene un subconjunto de elementos comunes de lenguaje de programación de software como C o Java. Además, hemos agregado elementos de álgebra lineal y operaciones específicas de los modelos predictivos. Estas adiciones permiten construir árboles de sintaxis abstracta más simples para la representación intermedia de modelos predictivos.

3.3.2 Modelos lineales generalizados (GLM)

En esta sección, se presenta una descripción de los modelos lineales generalizados (*Generalized Linear Model* o GLM). También se describen los elementos que definen un modelo lineal generalizado y las ecuaciones detrás de la función de predicción. En pocas palabras, el modelo lineal generalizado es una técnica de aprendizaje supervisado que incluye diferentes tipos de modelos de regresión lineal. El propósito de hacer una regresión es predecir el valor de una o más variables continuas denominadas objetivos (t) dado el valor de un vector dimensional p de variables de entrada, también conocidas como predictores.

En un modelo lineal generalizado, el valor observado de la variable dependiente y para la observación i se modela como una función lineal de las variables independientes $\{x_1, x_2, \ldots, x_p\}$. Los modelos lineales generalizados se construyen seleccionando una familia exponencial de distribución de probabilidad y una función de enlace. El modelo lineal generalizado es una generalización de modelos de regresión lineal ordinarios. Por lo tanto, se puede usar cualquier distribución que pertenezca a la familia exponencial. Esto incluye muchas distribuciones bien conocidas como binomial, gaussiano, Poisson y gamma. La función de enlace está relacionada con el valor esperado de la respuesta y a los predictores $\{x_1,\ ldots, x_p\ \}$. La función de enlace proporciona la relación de la media de la función de distribución y el valor del predictor lineal. La selección de una función de enlace depende del tipo de datos y su compatibilidad con la distribución familiar. El predictor lineal incorpora la información sobre las variables independientes en el modelo [87].

La operación principal del algoritmo de predicción de un modelo lineal generalizado es el producto de punto entre el vector predictor de coeficientes y un vector de variables de entrada de una observación. El producto punto es una operación algebraica que toma dos vectores y devuelve un escalar. El producto punto de dos vectores $\vec{A}$ y $\vec{B}$ de longitud n,

está definido por 3.1.

$$\vec{A} \cdot \vec{B} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \tag{3.1}$$

El modelo lineal generalizado es un enfoque unificado para el modelado. Después de construir un modelo lineal generalizado, contiene al menos los siguientes elementos; 1) información de las variables independientes como nombre y tipo, 2) un vector de coeficientes $\vec{\beta}$, 3) el valor del *intercept* α (generalmente conocido como sesgo), y 4) la función de enlace $g()$. Independientemente de la distribución seleccionada de la familia exponencial, la función de predicción se puede definir en términos de álgebra lineal. La Ecuación 3.2 define la función de predicción completa de un modelo lineal generalizado. Al resultado del producto punto entre el vector de entrada ($\vec{X}$ y los coeficientes ($\vec{\beta}$), se agrega un valor constante conocido como sesgo (α). El sesgo es el valor de la variable objetivo cuando todas las variables independientes son cero. Posteriormente, la función de enlace se aplica al resultado del pr esta operación.

$$prediction = g(\vec{X} \cdot \vec{\beta} + \alpha) \tag{3.2}$$

En el código fuente generado por el compilador, se crean dos funciones. Una función devuelve un valor en la escala de las predicciones lineales (función *link*) Una segunda función devuelve los valores en la escala del valor de respuesta (*i.e.,* aplica la función de enlace), y se conoce como la función *response*. Estas funciones se crean de la siguiente manera. El tipo de retorno de ambas funciones es un valor de coma flotante de doble precisión. La información de las variables independientes y su tipo se utilizan para definir los parámetros de las funciones. Si un parámetro es categórico, se crea una variable binaria proxy para cada categoría. Para la función *link*, se genera el código fuente para implementar la ecuación 3.3. Para la función *response*, se genera el código fuente para implementar la ecuación 3.4. Las funciones de enlace implementadas (junto con su descripción matemática) en el com-

pilador multi-destino propuesto se describen en la Tabla 3.2. En esta tabla, y es el resultado del uso de la ecuación 3.3 [88].

$$y = \vec{X} \cdot \vec{\beta} + \alpha \tag{3.3}$$

$$response = g(y) \tag{3.4}$$

El Algoritmo 1 muestra un ejemplo de una función de predicción GLM con una función de enlace logit. La línea 2 del algoritmo 1 solo es necesaria cuando el vector de entrada contiene valores categóricos; esta línea se reemplaza de la siguiente manera. Se define un nuevo vector I', este vector contiene todos los elementos numéricos anteriores del vector de entrada original I. Además, para cada categoría de cada valor categórico del vector de entrada I, se agrega un nuevo elemento con un valor igual a 0 al vector I'. Cuando se pasa un nuevo vector de entrada a la función, el valor 1 se asigna al elemento del vector I' correspondiente a la categoría de la entrada. Este proceso se realiza para cada valor categórico del vector de entrada. La información de los valores categóricos se encuentra en la descripción del modelo. El diseño de la plantilla de un GLM se puede ver en el ejemplo del árbol de sintaxis abstracta presentado en la Figura 3.4. En este árbol, los nodos en blanco son fijos, mientras que los nodos en gris son cambiados con la información de un modelo concreto. La operación principal está representada por el nodo con una leyenda *DOT*, que indica un producto de punto entre los vectores X e Y.

3.3.3 Redes neuronales artificiales

El modelo de red neuronal artificial (ANN) está inspirado en redes neuronales biológicas; Es un intento de representar una forma matemática de procesamiento de información por sistemas biológicos. Las ANN son modelos eficientes para el reconocimiento de patrones estadísticos y se pueden usar para regresión y clasificación [89]. Una red neuronal

Algorithm 1: Un ejemplo de un algoritmo de predicción de modelo lineal generalizado con una función de enlace logit. Este ejemplo implementa las ecuaciones 3.3 y 3.4.

Input: A feature vector $\vec{I} \leftarrow \{i_1, i_2, \ldots, i_n\}$ to score
Output: A numerical value

$\vec{\beta} \leftarrow$ Un vector de coeficientes
$\vec{I'} \leftarrow toNumerical(\vec{I})$ `/* Convierte valores categóricos a numéricos (opcional) */`
$\alpha \leftarrow$ Un valor numerico que representa el *intercept* o bias
$y \leftarrow 0$
$y \leftarrow (\vec{I'} \cdot \vec{\beta}) + \alpha$
return $1/(1 + exp(-y))$

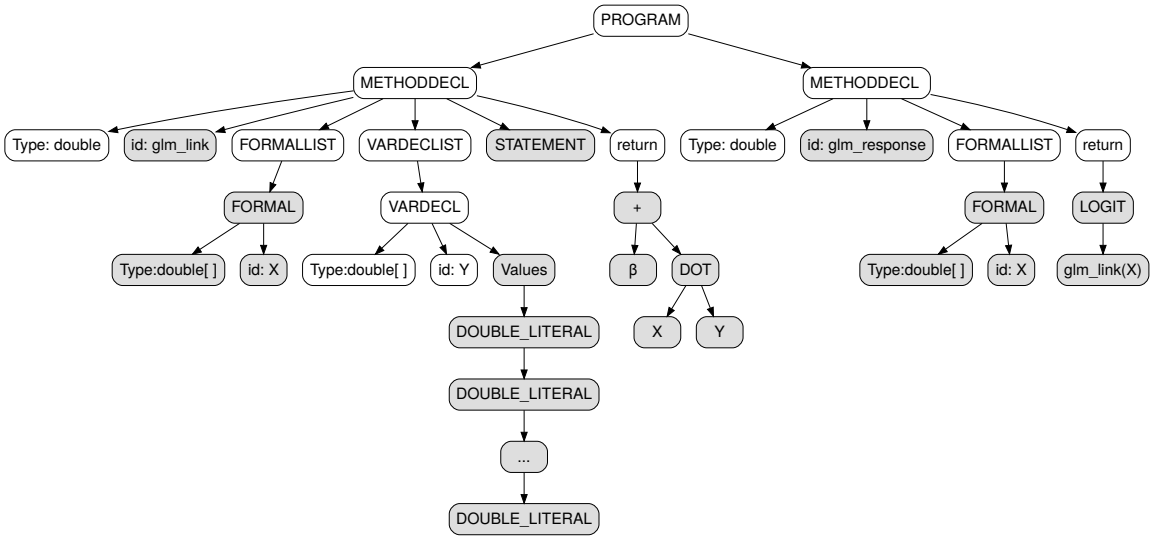


Figura 3.4: Ejemplo de una plantilla de árbol de sintaxis abstracta para una representación intermedia de un modelo lineal generalizado.

artificial se define por un grupo de unidades conectadas llamadas neuronas. Desde una perspectiva biológica, una neurona es una célula en el cerebro. La función principal de una neurona es la recolección, procesamiento y diseminación de señales eléctricas. Las redes para tales neuronas proporcionan la capacidad de procesamiento de información de los cerebros biológicos.

La Figura 3.5 muestra una representación gráfica de una neurona artificial [10]. Dada una neurona N que tiene un conjunto de enlaces entrantes E de un conjunto de neuronas A. Cada neurona en A está conectada a N a través de un enlace único en E. Cada neurona

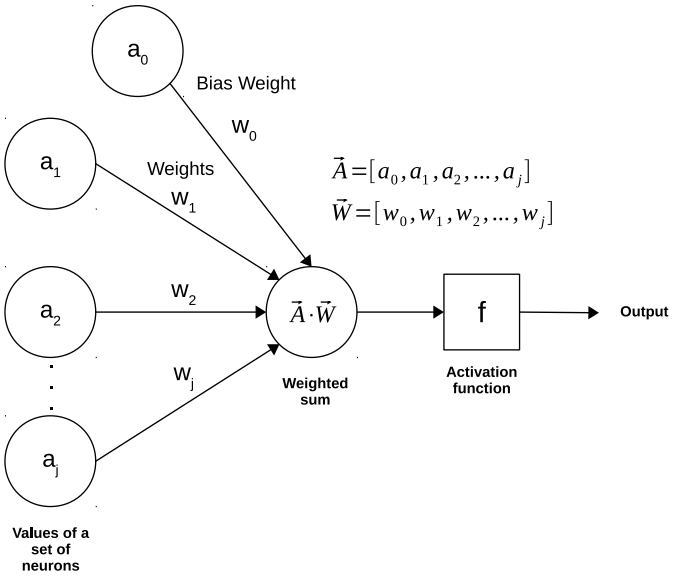
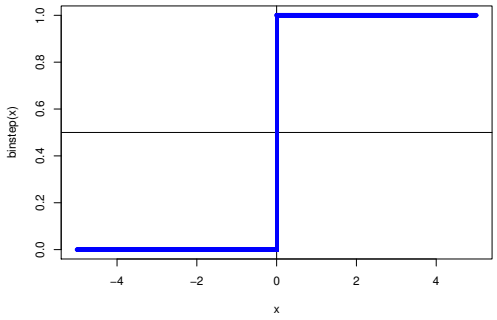


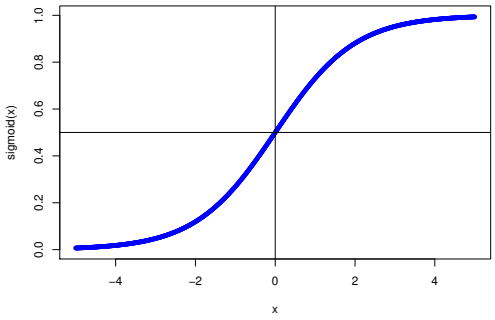
Figura 3.5: Representación gráfica de una neurona artificial.

tiene un valor asociado y cada enlace tiene un peso asociado que indica la importancia de una conexión entre dos neuronas. Por lo tanto, a_j está conectado a N a través del borde j-ésimo e_j. También se define una neurona de sesgo (*bias*) con un valor igual a uno y un enlace con un peso asociado. Todos los valores de salida de las neuronas en A se pueden definir como los valores de entrada de la neurona N. Lo llamamos el vector de entrada $\vec{A}$. De manera similar, todos los pesos asociados de los enlaces entrantes de la neurona N se pueden representar mediante un vector de pesos $\vec{W}$. El valor de la neurona N se obtiene de la siguiente manera. Primero, se calcula un producto de punto entre $\vec{A}$ y $\vec{W}$. Luego, al resultado del producto punto se aplica una función de activación.

En una red neuronal artificial, las neuronas se organizan en capas. La secuencia de esas capas define el orden en que se calculan los valores de las neuronas. Comúnmente, una red neuronal se compone de una capa de entrada, n capas ocultas y una capa de salida. Todas las funciones de activación de las neuronas se ejecutan antes de que la señal pase a la siguiente capa. El propósito de la función de activación es introducir la no linealidad en el valor de salida de una neurona. La función de activación hace que la neurona sea capaz de aprender y realizar tareas más complejas que un modelo de regresión lineal. Cuando se dan



(a) Binary step function.



(b) Sigmoid function.

Figura 3.6: Ejemplos de funciones de activación utilizadas en modelos de redes neuronales artificiales.

las entradas *correctas*, queremos que el valor de salida esté cerca de 1, mientras que cuando se dan las entradas *incorrectas* queremos que el valor de salida de una neurona esté cerca de 0. Hay diferentes funciones de activación utilizadas, las Figuras 3.6a y 3.6b muestran un ejemplo de funciones de activación paso binario y sigmoide respectivamente. La elección de la función de activación está determinada por la naturaleza de los datos y la distribución de las variables objetivo. La Tabla 3.3 muestra una lista de las funciones de activación más comunes y su representación matemática.

La Figura 3.7 muestra un ejemplo simple de una red neuronal multicapa. Cada capa está compuesta por un conjunto de nodos (neuronas). Los bordes representan la conexión entre las neuronas y la flecha del borde define la dirección del flujo de la señal durante la propagación. La base de cada cálculo neuronal se puede expresar como un producto puntual. Por lo tanto, podemos expresar un cálculo neuronal en términos de álgebra lineal como se muestra en la Ecuación 3.5, la función de activación $g()$ se aplica al resultado de un producto de punto entre $\vec{W}$ y $\vec{A}$.

$$neuron = g(\vec{W} \cdot \vec{A}) \tag{3.5}$$

El proceso de generación de una predicción con un modelo de red neuronal artificial es

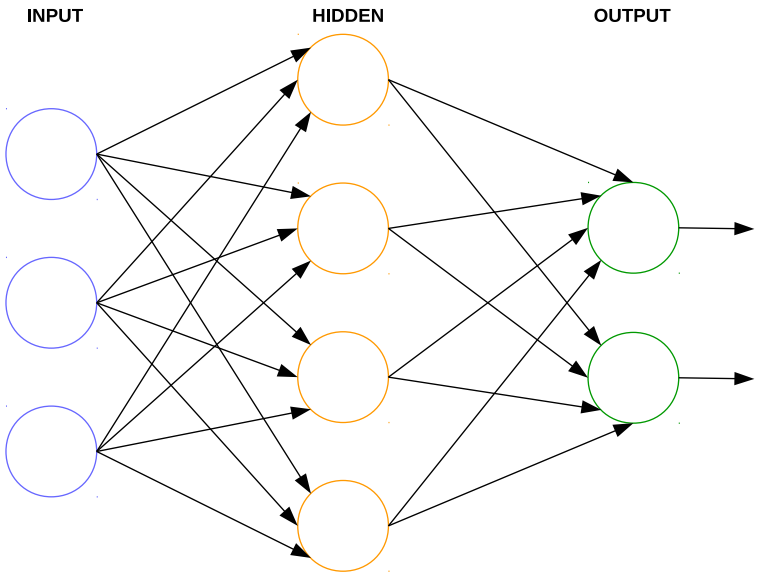


Figura 3.7: Ejemplo gráfico de una red neuronal multicapa.

el siguiente. Denominemos un modelo de red neuronal de n capas L, donde l_0 es la capa de entrada y l_n es la capa de salida. Cada capa puede contener un número diferente de neuronas. Una neurona se identifica por su índice de capa y un índice único dentro de esa capa. Los valores de una observación en la que queremos obtener una predicción se asignan a las neuronas en la capa de entrada. Para cada neurona de las capas siguientes, se calcula un producto de puntos entre los valores calculados por las neuronas en una capa anterior conectada a la neurona actual y los pesos asociados de cada conexión. Luego, se aplica una función de activación a cada resultado. Este proceso se realiza en orden, para cada capa, hasta que se procesen las neuronas de la capa de salida. El cálculo de una red neuronal se puede definir en términos de la ecuación 3.6. Donde l es el índice de capa y j es el índice dentro de la capa l de la neurona que queremos calcular. Los valores de entrada de $neuron_j^l$ están representados por $\vec{A}_j^{l-1}$ (los valores asociados de las neuronas conectadas de una capa anterior). Los pesos asociados de los bordes entre las neuronas de una capa anterior y la neurona actual están representados por $\vec{W}_j^l$. La función de activación está representada por $g()$. La capa de salida puede estar compuesta por una o más neuronas, y su resultado se

Algorithm 2: Pseudocódigo para implementar un algoritmo de predicción de un modelo de red neuronal.

Input: Un vector $\vec{I}$ representando una observación
Output: Un vector con los valores de las neuronas de la capa de salida

$G \leftarrow$ Una estructura de datos (gráfo dirigido) representando una red neuronal
$G_{0_{\vec{A}}} \leftarrow \vec{I}$ `/* capa de entrada */`
for ***cada capa*** $L \in G$ **do**
  for ***cada neurona*** $u \in L$ **do**
    `/*` $\vec{A}$ `es el vector de entrada y` $\vec{W}$ `es el vector de pesos de la neurona actual */`
    $u_{val} \leftarrow u_{\vec{W}} \cdot u_{\vec{A}}$
    $u_{val} \leftarrow g(u_{val})$
  end
end
return $G_{n_{\vec{A}}}$ `/* capa de salida */`

interpreta en términos de la variable objetivo.

$$neuron_j^l = g(\vec{W}_j^l \cdot A_j^{\vec{l-1}}) \tag{3.6}$$

El Algoritmo 2 muestra un pseudocódigo genérico para generar una predicción con un modelo de red neuronal artificial. La función recibe un vector de entrada I, que representa una observación, y G un gráfico acíclico dirigido que representa la estructura de una red neuronal. Los valores de I se asignan a las neuronas en la capa de entrada. Para procesar cada neurona en la red, se requieren dos ciclos anidados. El ciclo externo itera a través de cada capa, mientras que el ciclo interno itera sobre cada neurona de la capa actual. Para cada neurona, se calcula un producto punto entre el vector de entrada (valores de las neuronas de una capa anterior) y el vector de pesos (valores de los pesos asociados a las conexiones). Al resultado de la operación anterior se aplica una función de activación; dicha función de activación se define durante el entrenamiento del modelo. Finalmente, se devuelven los valores de las neuronas de la capa de salida. Las redes neuronales se pueden usar para regresión o clasificación.

La estructura general de un algoritmo de predicción para un modelo de red neuronal

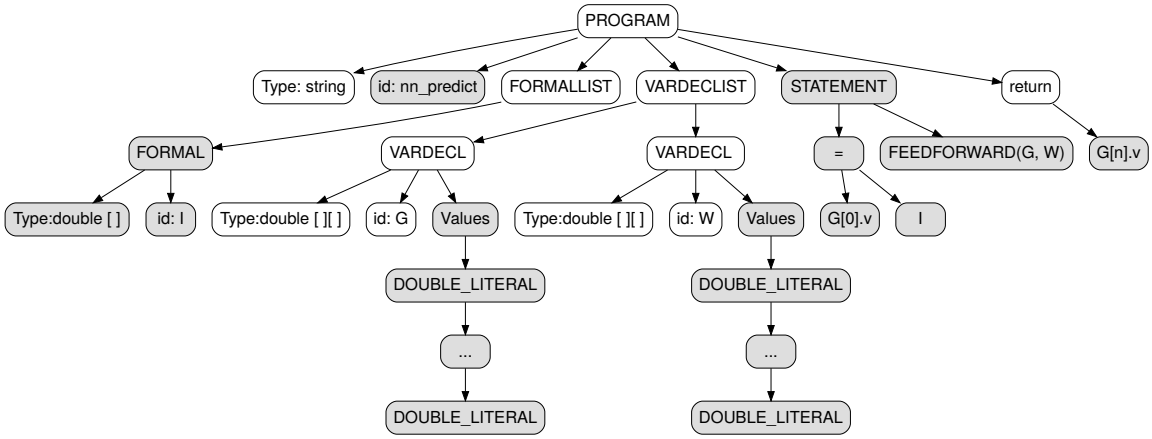


Figura 3.8: Ejemplo de una plantilla AST de un modelo de red neuronal.

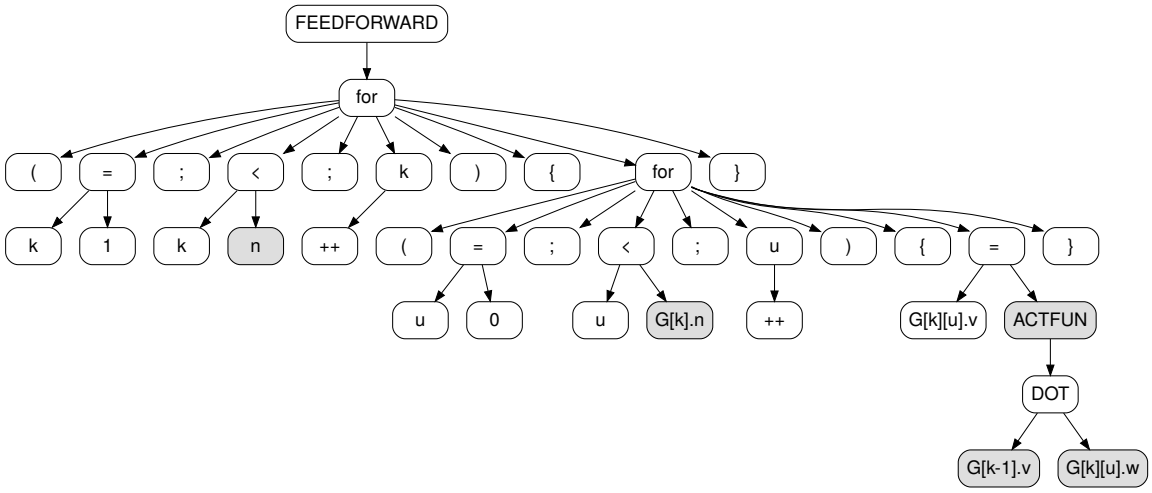


Figura 3.9: Ejemplo de una plantilla de AST del cómputo *feed-forward* de un modelo de red neuronal.

permite construir una plantilla para ser utilizada por el compilador. Concretamente, lo que cambia en una función de predicción de una red neuronal es la estructura de la red, los pesos, las funciones de activación, las entradas y las salidas. Sin embargo, la forma en que se calcula el valor de salida de la red es siempre la misma. En la Figura 3.8 se puede ver la estructura de una plantilla, un árbol de sintaxis abstracta de un modelo de red neuronal. Por cuestiones de legibilidad, la Figura 3.9 muestra la estructura detallada del cómputo de la red (*feed-forward*, es decir, los cálculos realizados para cada neurona. Los nodos cuyo fondo es blanco son fijos, mientras que los nodos de color gris son específicos del modelo.

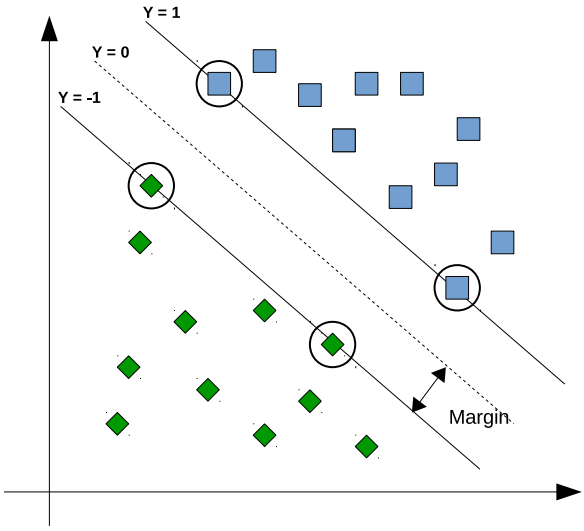


Figura 3.10: Ejemplo de un problema separable de dos clases en un espacio bidimensional, los vectores de soporte están marcados con un círculo y definen el margen máximo entre las dos clases.

3.3.4 Máquinas de vectores de soporte

Una máquina de vectores de soporte (SVM) es un método de aprendizaje automático supervisado para resolver problemas de clasificación y regresión. El SVM estándar es un clasificador binario lineal. El enfoque SVM introduce el concepto de margen, que se define como la distancia perpendicular entre el límite de decisión y el punto de datos más cercano. Un ejemplo de un espacio SVM bidimensional se muestra en la Figura 3.10. Maximizar el margen lleva a una división que permite elegir el límite de decisión. La ubicación de este límite está determinada por un subconjunto de los puntos de datos, que se denominan vectores de soporte.

Para los casos en que las clases no son linealmente separables, la técnica de SVM propone el uso del núcleo (*Kernel*). Esta propuesta de núcleo requiere que el espacio dimensional finito original se asigne a un espacio de dimensiones superiores, para permitir la separación más fácil en ese espacio. Esto significa que un clasificador SVM construye un hiperplano o un conjunto de hiperplanos en un espacio de alta dimensión. Un hiperplano óptimo se define como la función de decisión lineal con un margen máximo entre los vectores de las

dos clases [90]. Para mantener una carga computacional razonable, dicho mapeo utilizado por SVM está diseñado para garantizar que los productos de puntos se puedan calcular fácilmente en términos de las variables en el espacio original, definiéndolos en términos de una función del núcleo $k(\vec{x}, \vec{y})$ seleccionado para adaptarse al problema. Una vez que se entrena una SVM, contiene los llamados vectores de soporte $\vec{S}_i$, los coeficientes α y b, que son parámetros numéricos determinados por el algoritmo de entrenamiento SVM. La función de predicción se puede expresar usando álgebra lineal como se indica en la Ecuación 3.7. En esta ecuación, $\vec{x}$ es el vector de entrada n-dimensional que representa una observación y $\vec{x_i}$ es el vector de soporte i-ésimo. Cada tipo de función Kernel SVM tiene su propia definición. La Tabla 3.4 muestra las funciones de kernel más populares utilizadas por las SVM.

$$d(\vec{x}) = \sum_{i=1}^{m} \alpha_i K(\vec{x_i}, \vec{x}) + b \tag{3.7}$$

Con la excepción de radial, las funciones *kernel* de SVM listados en la Tabla 3.4 realizan una serie de productos de puntos entre el vector de entrada y todos los vectores de soporte. La operación principal de la ecuación para el núcleo radial es una norma al cuadrado, que se define en la Ecuación 3.8. La norma al cuadrado se puede reemplazar por la ecuación 3.9. Este reemplazo presenta una ventaja en términos de cálculos requeridos de la siguiente manera. Todas las operaciones de la ecuación 3.8 deben calcularse en tiempo de ejecución, mientras que la ecuación 3.9 reduce la cantidad de cálculos realizados en tiempo de ejecución de la siguiente manera; 1) el cálculo del auto producto punto del vector de entrada $\vec{x}$ se hace solo una vez en tiempo de ejecución, 2) cada auto producto punto de cada uno de los vectores de soporte $\vec{z}$ se pueden hacer en tiempo de compilación porque los valores son conocidos y están definidos en la descripción del modelo, y 3) en tiempo de ejecución, el producto punto entre la entrada $\vec{x}$ y el vector de soporte $\vec{z}$ se calcula y multiplica por dos. Con esta modificación, todos los tipos de *kernel* de SVM calculan un

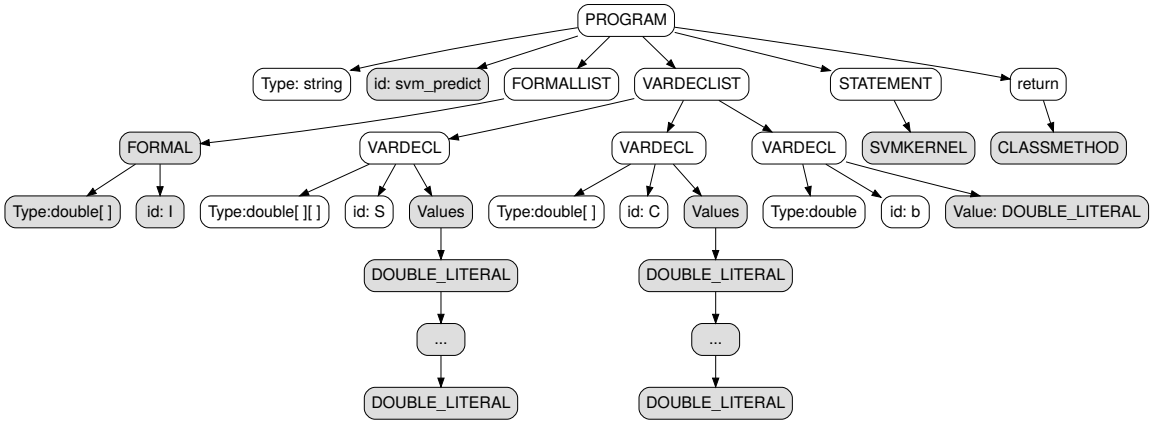


Figura 3.11: Árbol de sintaxis abstracta de la plantilla de un modelo de máquina de vectores de soporte.

producto de puntos entre el vector de entrada y cada vector de soporte del modelo SVM.

$$
\begin{aligned}
||\vec{x} - \vec{z}||^2 &= (\sqrt{(x_0 - z_0)^2 + \ldots + (x_n - z_n)^2})^2 \\
||\vec{x} - \vec{z}||^2 &= (x_0 - z_0)^2 + \ldots + (x_n - z_n)^2
\end{aligned}
\tag{3.8}
$$

$$
||\vec{x} - \vec{z}||^2 = (\vec{x} \cdot \vec{x}) - 2(\vec{x} \cdot \vec{z}) + (\vec{z} \cdot \vec{z}) \tag{3.9}
$$

Con la definición del cómputo principal de un modelo SVM como una operación de álgebra lineal, podemos definir una función de predicción. Se muestra un pseudocódigo de un modelo SVM de clasificación binaria en el Algoritmo 3. Además, la estructura general de un modelo SVM se muestra en la Figura 3.11. En esta figura, se presenta un ejemplo de una plantilla del árbol de sintaxis abstracta de una SVM. Los nodos cuyo color de fondo es blanco son fijos. Esto significa que estos nodos permanecen constantes en cada representación intermedia. Los nodos con fondo de color gris cambian con los valores específicos de un modelo. Los vectores de soporte, los coeficientes, el *kernel* y el método de clasificación es lo que cambia, el resto de la estructura del árbol permanece igual. Además, la Figura 3.12 presenta un ejemplo de una plantilla de AST de un núcleo lineal para ser utilizada por un modelo SVM.

El modelo de máquina de vectores de soporte es básicamente un clasificador binario.

Algorithm 3: Pseudocódigo para implementar un algoritmo de predicción de un SVM para la clasificación binaria.

Input: Un vector $\vec{I} \leftarrow \{i_1, i_2, \ldots, i_n\}$ de valores numéricos
Output: Una cadena que contiene la predicción del modelo SVM

$\vec{S} \leftarrow$ una matriz de vectores de soporte
$\vec{C} \leftarrow$ un vector de coeficientes
$R \leftarrow 0$
for $k \leftarrow 0$ ***to*** m **do**
    $R \leftarrow R + K(\vec{I}, \vec{s_k}) \times c_k$
end
$R \leftarrow R + b$
if $R > 0$ **then**
    return "A"
else
    return "B"
end

Sin embargo, las SVM también se pueden aplicar a problemas de clasificación múltiple. Existen varios métodos para combinar múltiples modelos SVM binarios para extender el número de clases que pueden clasificar. Un método popular para SVM multi-clase es *one vs. all*. Suponiendo que un problema de clasificación tiene N diferentes clases, el método *one vs. all* entrenará a un clasificador por clase. El clasificador para la clase i asumirá que las observaciones que pertenecen a la clase i son positivas y el resto como negativas. De los clasificadores entrenados, la predicción se obtiene del clasificador que tenga el puntaje más alto. Otro método multi-clase para SVM es *one vs. one*, donde se debe entrenar un clasificador separado para cada par de clases diferente. Asumiendo N como el número de clases, entonces $(N(N-1))/2$ modelos SVM deben ser entrenados para el método *one vs. one*. Para obtener la predicción, se deben calcular todos los modelos y cada modelo obtiene una clase (predicción). De esta lista de clases, la clase que tenga más apariciones en los resultados de los modelos se convierte en la predicción final.

La descripción de un SVM multiclase contiene más de un modelo de SVM, dado que todos los SVM *kernel* mencionados en la Tabla 3.4 realizan productos punto entre el vector de entrada y los vectores de soporte, podemos definir un algoritmo que realice ese opera-

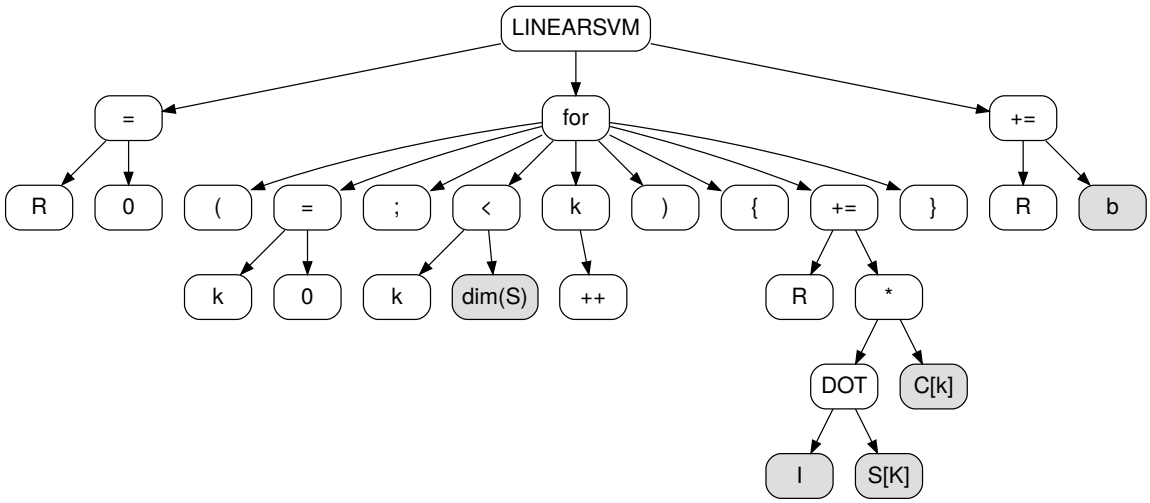


Figura 3.12: Árbol de sintaxis abstracta de la plantilla de un *kernel* lineal de un modelo de máquina de vectores de soporte.

ción y almacenar los resultados en un vector intermedio. En el Algoritmo 4 se presenta un pseudocódigo para la implementación de un SVM multiclase utilizando la técnica *one vs. one*. Funciona de la siguiente manera. Primero, todos los productos punto entre el vector de entrada y los vectores de soporte se calculan y los resultados se almacenan en el vector $\vec{R}$. El vector $\vec{Z}$ contiene el tamaño de cada modelo SVM y el vector $\vec{X}$ contiene los índices de los vectores de soporte de cada modelo de SVM construido para cada par de clases. Por lo tanto, los índices en $\vec{X}$ se utilizan para obtener los valores del producto de punto almacenado en $\vec{R}$ para su modelo SVM correspondiente. Luego, se realizan el resto de los cálculos específicos del *kernel*. Cada resultado de una función del *kernel* se multiplica por su coeficiente correspondiente. Los resultados de esas multiplicaciones se acumulan en una variable. Luego, se agrega el sesgo y las puntuaciones individuales para cada modelo SVM se almacenan en el vector $\vec{Q}$. Cada modelo SVM tiene un valor de sesgo definido por el algoritmo de entrenamiento. Finalmente, la clase que tiene más votos es la que devuelve la función de predicción.

Algorithm 4: Implementación de un algoritmo de predicción de un SVM multiclase utilizando la técnica *one vs. one*.

Input: Un vector $\vec{I} \leftarrow \{i_1, i_2, \ldots, i_n\}$ de valores numéricos
Output: Una cadena que contiene la etiqueta con la predicción del modelo SVM

$\vec{S} \leftarrow$ una matriz de vectores de soporte
$\vec{C} \leftarrow$ un vector de coeficientes
$\vec{X} \leftarrow$ un vector de índices de cada modelos SVM
$\vec{Z} \leftarrow$ un vector indicando la cantidad de los vectores de sporte para cada modelo SVM
$\vec{W} \leftarrow$ un vector de etiquetas representando a las clases
$\vec{T} \leftarrow$ un vector indicando las posiciones de la clase *target*
$\vec{A} \leftarrow$ un vector indicando las posiciones de la clase *alternate target*
$\vec{B} \leftarrow$ un vector de coeficientes absolutos de cada modelo SVM
$\vec{R} \leftarrow$ un vector que contiene los resultados de los productos punto
for $k \leftarrow 0$ ***to*** $|\vec{S}|$ **do**
    $R_k \leftarrow \vec{I} \cdot \vec{s_k}$ `/* calcular los productos punto */`
end
$\vec{Q} \leftarrow$ un vector para almacenar los resultados finales de cada modelos SVM
for $j \leftarrow 0$ ***to*** $|\vec{B}|$ **do**
    $Q_j \leftarrow 0$
    for $l \leftarrow 0$ ***to*** Z_j **do**
        $Q_j \leftarrow Q_j + K'(R_{X_l}) \times C_l$ `/* K' se refiere al resto de operaciones específicas de una función kernel */`
    end
    $Q_j \leftarrow Q_j + B_j$
end
$\vec{V} \leftarrow$ un vector que contiene los votos de cada clase
for $j \leftarrow 0$ ***to*** $|\vec{B}|$ **do**
    `/* etiqueta en` T `contra etiqueta en` A `*/`
    if $Q_j > 0$ **then**
        $V_{T_j} \leftarrow V_{T_j} + 1$
    else
        $V_{A_j} \leftarrow V_{A_j} + 1$
    end
end
$M \leftarrow 0$ `/* almacena el índice de la etiqueta más votada en el vector` W `*/`
for $j \leftarrow 1$ ***to*** $|\vec{W}|$ **do**
    if $V_M < V_j$ **then**
        $M \leftarrow j$
end
return W_M

3.4 Back-End

El *back-end* del compilador se encarga de recibir una representación intermedia y generar el código fuente correspondiente en un lenguaje de programación destino. Dado el diseño del compilador, podemos construir varios módulos independientes para generar código para diferentes lenguajes de programación destino. En un compilador *clásico*, el *back-end* realiza varias tareas, como la selección de instrucciones, la programación de instrucciones y la asignación de registros para generar el código máquina que se ejecutará. En el caso del compilador propuesto, nuestro objetivo es generar código fuente en un lenguaje de programación. El código fuente generado debe implementar la función de predicción del modelo predictivo dado. Este código fuente automatizaría la tarea de implementación de modelos predictivos. Además, el código generado está listo para ser compilado (por un compilador de lenguaje de programación específico) para generar un programa que se ejecute de manera eficiente y directa en una máquina destino o una máquina virtual.

Los *back-end* desarrollados para el compilador propuesto requieren un módulo generador de código. Aunque también proponemos implementar un módulo optimizador de acuerdo con el idioma destino. El generador de código asigna las operaciones de IR a las operaciones del lenguaje destino. Este módulo maneja los detalles de generar una representación concreta para las operaciones de representación intermedia. Se deben tomar decisiones de diseño importantes porque generalmente hay más de una forma de implementar la operación de IR en el idioma de destino. Este módulo también puede dar formato y almacenar el código generado en un archivo. El generador de código recibe como entrada una IR de código, la IR tiene una estructura de árbol. Luego, se atraviesa el árbol y para cada operación de IR se genera una instrucción concreta en el lenguaje destino hasta que se visiten todos los nodos. El módulo optimizador es opcional y se encarga de transformar la representación intermedia en una versión optimizada de acuerdo con el idioma de destino. El optimizador se vuelve útil cuando el código generado se va a ejecutar en arquitecturas

de computación paralelas, como múltiples núcleos o GPU. Presentamos estas estrategias de optimización en el Capítulo 4.

PROGRAM	→	MAINCLASS CLASSDECL* | MAINCLASS CLASSDECL*
		| METHODDECL*
MAINCLASS	→	**class** id { CLASSDECL* METHODDECL* }
		| **package** STRING_LITERAL | **import** STRING_LITERAL
CLASSDECL	→	**class** id { VARDECL* METHODDECL* }
VARMOD	→	static | const | public | private | protected | long | short | unsigned | signed
VARDECL	→	VARMOD TYPE id; | VARMOD TYPE id = EXP;
METHODDECL	→	TYPE id (FormalList) { VARDECL STATEMENT return EXP;}
		| MODIFIER TYPE id (FormalList) { VARDECL STATEMENT return EXP;}
FORMALLIST	→	TYPE id FORMALREST*
FORMALREST	→	, TYPE id
TYPE	→	boolean | int | float | double | string | int [ ] | float [ ] | double [ ] | string [ ]
STATEMENT	→	{ STATEMENT* } | if (EXP) STATEMENT else STATEMENT
		| while (EXP) STATEMENT | id = EXP; | id[EXP] = EXP;
		| for (FORINIT; EXP ; FORUPDATE) STATEMENT | STATEMENTEXP
STATEMENTEXP	→	LINALG | LINKFUN | ACTFUN | SVMOP | TRANSFORMS | EXP
EXP	→	EXP op EXP | EXP [EXP] | id (EXPLIST) | id | STATEMENTEXP
		| id<<<EXP, EXP, EXP >>>(EXPLIST) | STRINGCOMPAREEXP
		| LITERALEXP
EXPLIST	→	EXP EXPREST* | λ
EXPREST	→	, EXP
LITERALEXP	→	INTEGER_LITERAL | FLOAT_LITERAL | DOUBLE_LITERAL
		| CHARACTER_LITERAL | STRING_LITERAL | true | false | null
OP	→	ASSIGNOP | INFIXOP | PREFIXOP | POSTFIXOP
ASSIGNOP	→	$= \mid + = \mid - = \mid * = \mid / =$
INFIXOP	→	$\mid \mid\mid \mid \&\& \mid \mid \mid == \mid != \mid < \mid > \mid <= \mid >= \mid << \mid >> \mid + \mid - \mid * \mid / \mid \%$
PREFIXOP	→	$++ \mid -- \mid ! \mid \tilde{} \mid + \mid -$
TYPE	→	boolean | double | identifier | integer | char | string | void
LINALG	→	DOT | NORM
LINKFUN	→	IDENT | LOGIT | PROBIT | CLOGLOG | LOG | SQRT | INVERSE | MU
ACTFUN	→	SIGMOIDNN | BINARYNN | LINEARNN | TANHNN | RELUNN
SVMOP	→	SVMKERNEL | CLASSMETHOD
SVMKERNEL	→	LINEARSVM | RBFSVM | POLYNOMIALSVM | SIGMOIDSVM
CLASSMETHOD	→	CMBINARY | CMOVSO
TRANSFORMS	→	NORMCONTINUOUS | NORMDISCRETE

Tabla 3.1: Gramática utilizada por la representación intermedia de código del compilador multi-destino propuesto.

Función de enlace	**Ecuación**
identity	$f(y) = y$
logit	$f(y) = \frac{1}{1+e^{-y}}$
probit	$f(y) = \Phi^{-1}(y)$
cloglog	$f(y) = 1 - e^{-e^{y}}$
log	$f(y) = e^{y}$
sqrt	$f(y) = \sqrt{y}$
inverse	$f(y) = y^{-1}$
$1/mu^2$	$f(y) = \Phi(\sqrt{\frac{\lambda}{x}}(\frac{x}{y} - 1)) + e^{\frac{2\lambda}{y}}\Phi(-\sqrt{\frac{\lambda}{x}}(\frac{x}{y} + 1))$

En donde Φ es la distribución normal estándar, λ es la varianza y x es la media.

Tabla 3.2: Lista de funciones de enlace implementadas en el compilador y su ecuación.

Función de activación	**Ecuación**
binary step	$f(y) = \begin{cases} 0 & \text{for } y < 0 \\ 1 & \text{for } y \geq 0 \end{cases}$
logistic/sigmoid	$f(y) = \frac{1}{1+e^{-y}}$
tanh	$f(y) = tanh(y) = \frac{2}{1+e^{-2y}} - 1$
identity	$f(y) = y$
exponential	$f(y) = e^{y}$
reciprocal	$f(y) = 1/y$
square	$f(y) = y^2$
Gauss	$f(y) = e^{-y^2}$
sine	$f(y) = sin(y)$
cosine	$f(y) = cos(y)$
Elliot	$f(y) = \frac{0.5(y)}{1+\|y\|} + 0.5$
arctan	$f(y) = 2(\frac{arctan(y)}{\pi})$
rectifier	$f(y) = \begin{cases} 0 & \text{for } y < 0 \\ y & \text{for } y \geq 0 \end{cases}$

Tabla 3.3: Lista de las funciones de activación más comunes utilizadas en modelos de redes neuronales y su ecuación.

Kernel Function	**Equation**
Lineal	$K(\vec{x}, \vec{z}) = \vec{x} \cdot \vec{z}$
Polinomial	$K(\vec{x}, \vec{z}) = (\gamma * (\vec{x} \cdot \vec{z}) + c)^d$
Radial	$K(\vec{x}, \vec{z}) = e^{-\gamma * \|\|\vec{x} - \vec{z}\|\|^2}$
Sigmoid	$K(\vec{x}, \vec{z}) = tanh(\gamma * (\vec{x} \cdot \vec{z}) + c)$

en donde γ, $coef0$ y d son determinados por el algoritmo de entrenamiento de SVM.

Tabla 3.4: Lista de las funciones *kernel* más utilizados por máquinas de vectores de soporte
.

CAPÍTULO 4
IMPLEMENTACIÓN DEL COMPILADOR MULTI-DESTINO

En este capítulo, se describe la implementación del compilador multi-destino para el despliegue de modelos predictivos. La descripción del diseño de la implementación se presenta en la Sección 4.1, mientras que el *front-end* y el *back-end* se describen en las Secciones 4.2 y 4.3, respectivamente.

4.1 Descripción general de la implementación

Dado que existe una amplia variedad de herramientas para construir modelos, para la implementación del compilador propuesto, optamos por un *front-end* para modelos descritos en PMML. Este *front-end* permite utilizar muchas herramientas analíticas capaces de exportar modelos predictivos a PMML. De esta manera, cubrimos una buena cantidad de herramientas de modelado. También hemos desarrollado *back-end* para esta implementación que cubren los lenguajes de programación C y Java. Se han desarrollado *back-end* adicionales al extender el *back-end* del lenguaje C proporcionando generación de código de funciones definidas por el usuario para sistemas gestores de bases de datos y generación de código para entornos de cómputo heterogéneos, como procesadores multi-core o unidades de procesamiento de gráficos (GPUs).

Para el diseño del compilador, utilizamos una técnica conocida como el patrón de diseño visitante [91]. Este patrón requiere un lenguaje de programación orientado a objetos; Optamos por el lenguaje Java. El patrón de diseño visitante es muy utilizado en la construcción de compiladores y permite definir nuevas operaciones en la estructura de un objeto sin cambiar la definición de la clase de los objetos. Este patrón tiene al menos las siguientes clases participantes: visitante, visitante concreto, visitable, visitable concreto y estructura de objeto. La clase visitante es una interfaz utilizada para declarar las operaciones de visita

para todo tipo de clases visitables. Las clases visitante concretas implementan los métodos declarados en la clase visitante. Cada visitante concreto realiza diferentes operaciones. La clase visitable es una clase abstracta que tiene un método *accept( )*, que es el punto de entrada para ser "visitado" por un objeto visitante concreto. Las clases visitables concretas implementan la clase visitable y definen un método concreto *accept( )*. La clase estructura de objeto contiene todos los objetos que se pueden visitar y proporciona un mecanismo para recorrer todos los elementos.

El diseño del compilador multi-destino propuesto utilizando el patrón de diseño visitante se presenta en el diagrama de clase de la Figura 4.1. En este diagrama, hay una clase cliente que usa otras clases en el proceso de compilación. Para el *front-end*, hemos desarrollado una clase que realiza el análisis léxico, el análisis sintáctico y utiliza las plantillas para generar una representación intermedia. Hemos definido una clase abstracta para las plantillas IR (clase de estructura de objeto) y clases concretas para cada uno de los tipos de modelos predictivos compatibles. Las plantillas contienen elementos de la gramática libre de contexto para construir un programa que implemente la función de predicción para el modelo predictivo actual. Hemos definido una clase abstracta para la gramática (clase visitable) y clases concretas (visitable concreta) para cada elemento que forma parte de la gramática. La clase de cliente también usa las clases de generadores de código que implementan la clase *Generator*. La clase Generator desempeña el papel del visitante. Esta interfaz contiene un conjunto de métodos sobrecargados denominados `visit( )` que recibe un elemento de la gramática como parámetro. Hay un método `visit( )` para cada elemento de la gramática. Los generadores concretos (visitantes de concreto) implementan la interfaz Generator. Por lo tanto, estos generadores concretos tienen métodos `visit( )` significativos que se encargan de generar el código equivalente de lo que están recibiendo como parámetros. Todas las clases que heredan de la clase Grammar tienen un método `accept( )` que recibe como parámetro un objeto generador.

Por ejemplo, supongamos que hemos traducido un archivo PMML a la IR utilizando

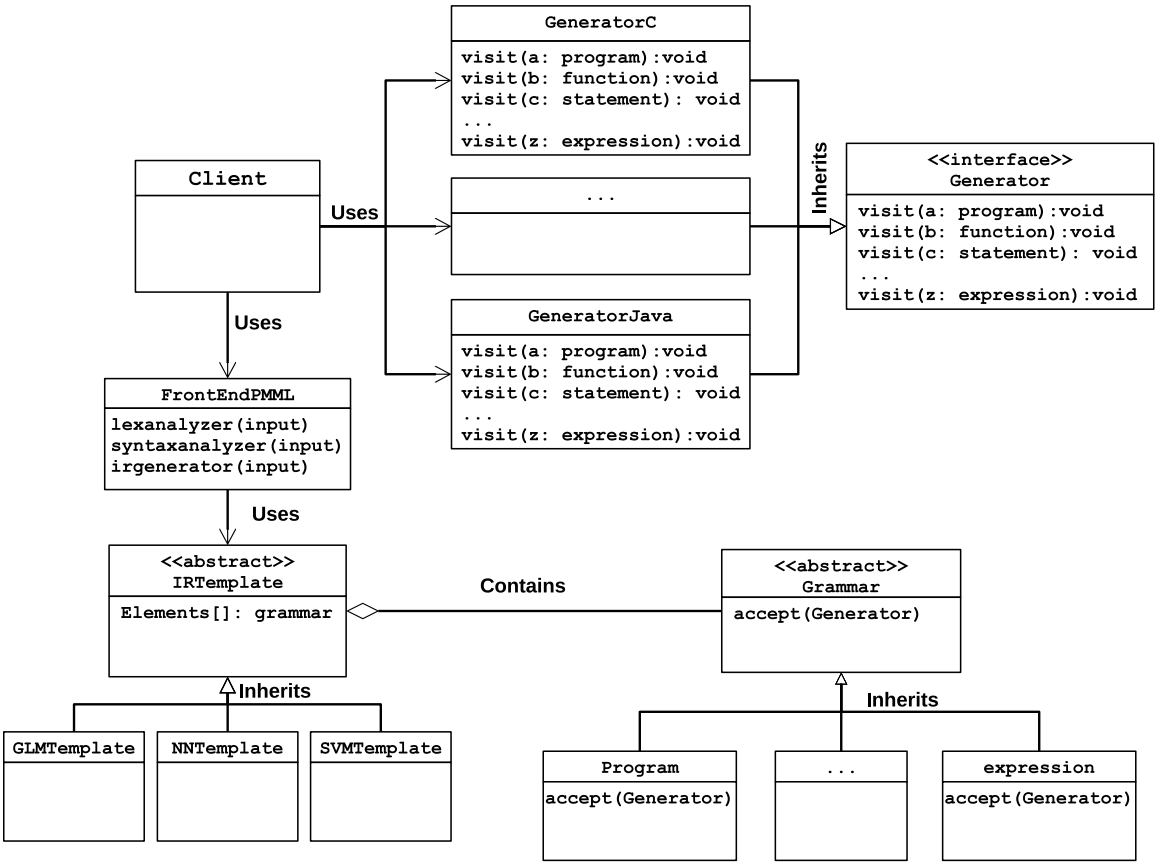


Figura 4.1: Diagrama de clases de la implementación del compilador multi-destino para el despliegue de modelos predictivos.

una plantilla y el idioma de destino es C. Además, la IR tiene como elemento raíz un objeto *Program* que se define en nuestro CFG. Desde nuestro cliente invocamos el método `accept( )` de *program* y pasamos como parámetro un objeto generador para C (*GeneratorC*). Esta acción activará el método `visit( )` del objeto *GeneratorC*. El método `visit( )` atraviesa todos los elementos de una regla de derivación. Si el elemento de una regla de derivación es terminal, se genera el código correspondiente. De lo contrario, se invoca el método `accept( )` y utilizamos el mismo objeto generador para el código C como parámetro. Este proceso atravesará toda la representación intermedia y generará el código correspondiente hasta que no haya elementos no terminales en la IR.

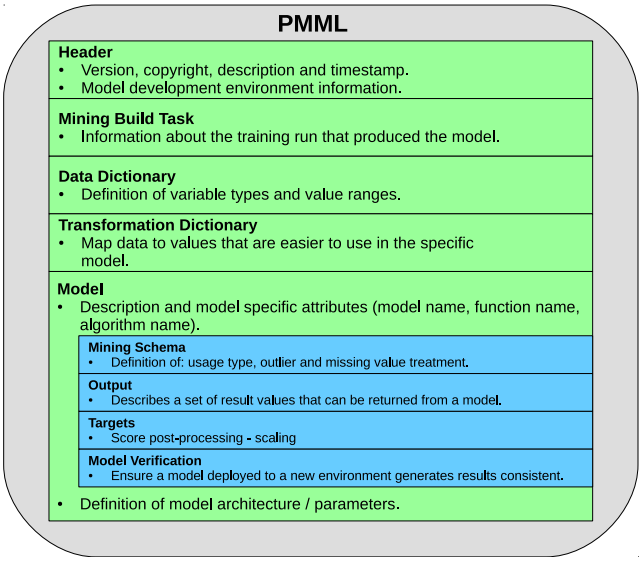


Figura 4.2: Descripción de la estructura general de un archivo PMML.

4.2 Front-End

Los modelos persistentes como los descritos en el formato PMML son generados por herramientas de modelado. Por lo tanto, no es necesario editar a mano el archivo que contiene un modelo; Es generado por computadora. Esto permite una simplificación de la tarea de verificación de sintaxis porque el proceso es de máquina a máquina. PMML proporciona un estándar abierto basado en lenguaje de marcado extensible (XML) para representar modelos predictivos y transformaciones básicas de datos. El estándar PMML tiene una estructura compuesta por muchos elementos que encapsula sus características principalmente a los datos de entrada, modelo y salidas. La Figura 4.2 muestra la estructura general de un archivo PMML, que contiene los siguientes elementos [17].

- Header. Este elemento contiene información general sobre el documento y la siguiente información sobre el modelo: derechos de autor, descripción, detalles de la aplicación utilizada para generar el PMML y sello de tiempo de su creación.
- Mining Build Task. Son valores XML que describen los parámetros de la tarea de capacitación que produjo la instancia del modelo están contenidos en este elemento.

Esta información no la necesita directamente un consumidor de PMML, pero puede ser útil para el mantenimiento y la visualización del modelo.

- Data Dictionary. El diccionario de datos contiene definiciones para todas las posibles variables utilizadas por los modelos. Define el tipo de las variables: continua, categórica u ordinal. Dependiendo de esta definición, los rangos de valores apropiados se definen como los tipos de datos de las variables.

- Transformation Dictionary. Este elemento contiene información para asignar los datos de entrada al formato o tipo que necesita el modelo. PMML define las siguientes transformaciones de datos simples.

    - Normalization. Esta transformación de datos asigna valores de entrada a números de valores específicos, generalmente al rango numérico de 0 a 1.
    - Discretization. En esta transformación de datos, las variables de entrada numéricas se asignan de valores continuos a discretos utilizando intervalos.
    - Value Mapping. Esta transformación de datos asigna un valor discreto a otro valor discreto enumerando los pares de valores. Esta lista es implementada por una tabla.
    - Functions. Esta opción permite especificar una función personalizada para transformar valores. La función puede tener uno o más parámetros.
    - Aggregation. En esta transformación de datos, los datos de entrada se agrupan por una de las variables. Luego, se puede aplicar una función agregada, como conteo, suma, promedio, mínimo o máximo.
    - Lag. Esta transformación solo tiene sentido si se ordenan los datos. Un valor de retraso se refiere como el valor de un campo de entrada dado a un número fijo de registros anteriores al actual.

- Model. Este elemento contiene la definición del modelo predictivo. Los modelos generalmente tienen un nombre de modelo, tipo de función (clasificación o regresión) y atributos específicos del modelo. La representación del modelo comienza con un esquema de minería, objetivos y continúa con la representación real del modelo.

    - Mining Schema. En este elemento se enumeran todas las variables utilizadas en el modelo. Puede ser un subconjunto de las variables definidas en el diccionario de datos. Define las variables predictoras y de destino (predichas) utilizadas en el modelo. También especifica cómo se tratan los valores especiales. Esto incluye el tratamiento de valores atípicos, la política de reemplazo del valor perdido o el tratamiento del valor perdido.
    - Output. Este elemento describe un conjunto de valores de resultados que se pueden devolver desde el modelo. Los elementos de salida especifican nombres, tipos y reglas para calcular características de resultado específicas.
    - Target. El elemento *target* proporciona una sintaxis común para todos los modelos. Los valores objetivo se derivan de una variedad de elementos en los modelos. Por ejemplo, las categorías destino de un modelo de regresión pueden especificarse mediante umbrales numéricos.
    - Model Verification. El elemento de verificación del modelo agrega un conjunto de datos de verificación al modelo. Es un mecanismo para proveedores y consumidores de modelos PMML para garantizar que un modelo implementado en un nuevo entorno genere resultados consistentes con el entorno donde se desarrolló el modelo.
    - Model Element. Este elemento incluye detalles específicos del modelo predictivo. Por ejemplo: una red neuronal de alimentación de múltiples capas se define por las capas, neuronas, conexiones, funciones de activación y pesos (coeficientes).

El grupo Data Mining es un consorcio que desarrolla estándares de minería de datos y se encarga de liberar y mantener el estándar PMML. Se proporciona un esquema XML con cada versión de lanzamiento de PMML. Este esquema especifica cómo describir formalmente los elementos de un archivo PMML. Al tener el esquema XML, podemos usar herramientas existentes, como *Java Architecture for XML Binding* (JAXB) para traducir automáticamente un archivo XML a clases Java. Utilizamos JAXB para generar clases Java equivalentes a partir del esquema PMML y un módulo *marshalling*. El *marshalling* se encarga de leer un archivo PMML, validar que la entrada cumple con las reglas especificadas por el esquema XML y mapear la descripción del modelo en clases Java. A partir de ahí, podemos extraer información relevante del modelo predictivo y usar las plantillas definidas para generar una representación intermedia.

4.3 Back-End

En esta sección describimos los distintos *back-end*. En esta etapa, hemos seleccionado C y Java como nuestros principales lenguajes destino porque se encuentran entre los lenguajes de programación más populares para crear software. Por lo tanto, están presentes en muchos entornos operativos. Según el índice TIOBE [1], Java está en la primera posición y C está en la segunda posición como lenguajes de programación. El índice TIOBE es un indicador de la popularidad de los lenguajes de programación de software basados en diferentes estadísticas de los motores de búsqueda más populares. En este índice, C y Java han estado en las primeras posiciones durante muchos años. Ampliamos el *back-end* del lenguaje C para generar código para implementar modelos como funciones definidas por el usuario en los sistemas gestores de base de datos (DBMS). Además, también ampliamos el *back-end* del lenguaje C para generar código para ejecutar los algoritmos de predicción en entornos de núcleos múltiples utilizando OpenMP y en Unidades de Procesamiento Gráfico (GPU) utilizando CUDA C.

[1] `https://www.tiobe.com/tiobe-index/`, diciembre de 2019.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

double regression(double A, char B[], ..., double N){
  ...
  return prediction;
}

const char *  classification(double A, char B[], ..., double N){
  ...
  return prediction;
}
```

Listado 4.1: Ejemplo de prototipo de un código C generado.

4.3.1 Generación de código en el lenguaje C

C es un lenguaje orientado a funciones, el código C compilado se traduce de manera eficiente a las instrucciones de la máquina. Por lo tanto, es ampliamente utilizado en sistemas operativos, aplicaciones en general y sistemas embebidos. El código C compatible con el estándar se puede compilar en diferentes tipos de dispositivos y entornos operativos, ofreciendo una alta portabilidad. Las aplicaciones creadas con el lenguaje C se ejecutan como código máquina ejecutable nativo; Por lo tanto, las aplicaciones pueden ser muy eficientes. En muchos casos, se prefiere C sobre otros lenguajes de programación cuando el software a construir requiere un control muy estricto de la memoria, tiempos de respuesta bajos y alto rendimiento.

El compilador propuesto genera código C que contiene funciones listas para ser llamadas desde otro programa. El Listado 4.1 muestra un prototipo de ejemplo del código generado por el compilador propuesto usando C como lenguaje destino. Por defecto incluye las bibliotecas básicas y al menos una función. Esta función recibe, como parámetros, los valores del vector característico a procesar y devuelve un valor numérico (para regresión) o una cadena (para clasificación). Además, se crea un archivo *header* con los prototipos de las funciones en el código generado. Este archivo *header* permite crear un archivo de objeto para que la función se pueda compilar y vincular desde otros programas.

```
1  package test;
2
3  public class test_class{
4
5    public static double regression(double A, String B, ..., double N){
6      ...
7      return prediction;
8    }
9    public static String classification(double A, String B, ..., double N){
10     ...
11     return prediction;
12   }
13 }
```

Listado 4.2: Ejemplo de prototipo de un código generado por Java.

4.3.2 Generación de código en lenguaje Java

Java es un lenguaje de alto nivel orientado a objetos; su sintaxis se deriva de C. Las aplicaciones Java no se compilan en instrucciones de máquina sino en un lenguaje intermedio. Este lenguaje intermedio se conoce como *Bytecode* y puede ejecutarse en cualquier plataforma con una Java Virtual Machine (JVM) instalada. Muchos entornos de programación y entornos populares son compatibles con Java, por ejemplo: Spring, Android, Apache Hadoop, servidores web Java (Tomcat, JBoss, Glassfish), por nombrar algunos.

El compilador propuesto genera código Java listo para ser utilizado. Las funciones prototipo utilizadas son muy similares a las creadas para C. Sin embargo, Java es un lenguaje orientado a objetos y debe crearse una clase que contenga las funciones. Además, se debe especificar un paquete. El usuario puede pasar el nombre de la clase y el paquete al compilador o modificarlo manualmente. Un prototipo del código generado para Java se presenta en el Listado 4.2. La clase generada es pública por defecto. Las funciones tienen modificadores públicos y estáticos por defecto.

4.3.3 Generación de UDFs para sistemas gestores de bases de datos

Los datos pueden ser eficientemente almacenados, administrados y procesados dentro de los sistemas gestores de bases de datos. Los beneficios más importantes de los DBMS son la gestión robusta y eficiente de grandes conjuntos de datos y la independencia de datos

físicos que proporcionan, ya que permiten el almacenamiento y la recuperación de datos en términos de un modelo de datos abstracto que oculta detalles de almacenamiento de bajo nivel. Beneficios adicionales como control de acceso (seguridad), concurrencia, recuperación de fallas, disponibilidad y escalabilidad también están disponibles para los usuarios y las aplicaciones porque son partes integrales de un DBMS [92]. Además, los DBMS modernos pueden ampliar su rango de capacidades de procesamiento de datos al vincular funciones especializadas desarrolladas fuera del sistema. Estas funciones, conocidas como funciones definidas por el usuario (UDF), están escritas en lenguajes de programación de alto nivel como Java o C++. Los UDF se pueden usar para implementar funciones de preprocesamiento o modelos predictivos dentro del DBMS. Una vez que estas funciones se han vinculado al DBMS, se pueden usar fácilmente como parte de una instrucción SQL.

Al extender el lenguaje C, podemos generar código para implementar modelos como UDF dentro de un DBMS. Por lo general, una función de envoltura debe definirse como un intermediario entre el DBMS y la función que calcula la predicción. En esta etapa, hemos desarrollado dos generadores de código para apuntar a los siguientes DBMS; MySQL y Vertica.

MySQL. Este es un sistema de gestión de bases de datos de código abierto muy popular. Es muy utilizado por los sistemas web. Para crear una UDF para MySQL, se debe crear un *wrapper* para mapear los valores de entrada y salida de la función de predicción. En MySQL, estos *wrappers* se conocen como interfaces. La Figura 4.3 ilustra el proceso general de la implementación de una UDF. Primero, se debe declarar una función `init( )`, el nombre de ésta función se construye con el nombre de la función principal que se va a invocar seguido del texto `_init`. Esta función está a cargo de lo siguiente; 1) mapear los valores de MySQL a C, 2) validar que el número de parámetros es correcto, 3) asignar recursos y 4) opcionalmente señalar un error si se detecta. Luego, para cada fila se invoca una función principal de la UDF. Dentro de esta función principal se invoca la función de pre-

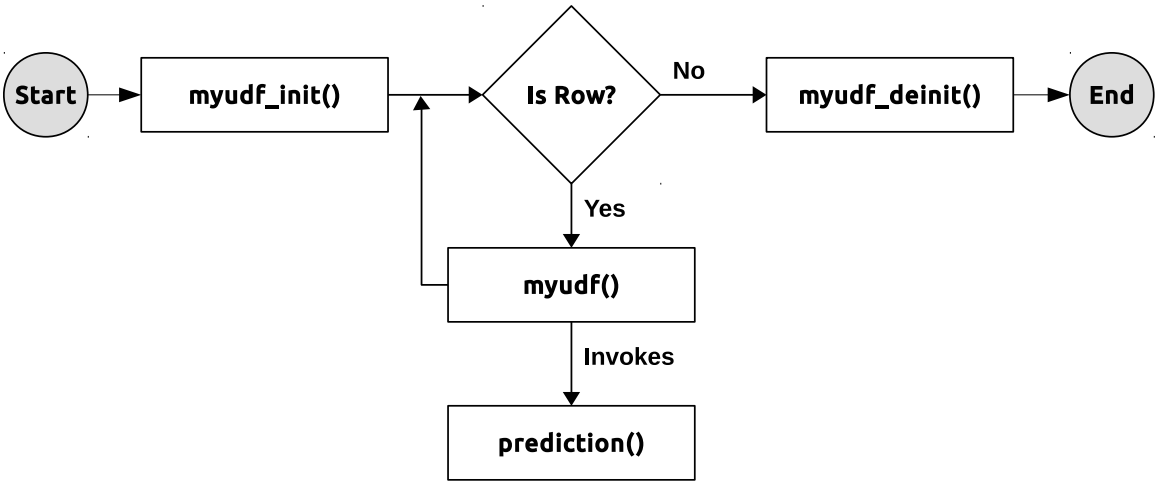


Figura 4.3: Descripción del funcionamiento de una UDF en MySQL.

```
1 CREATE FUNCTION function_name
2 RETURNS {STRING|INTEGER|REAL|DECIMAL}
3 SONAME shared_library_name;
```

Listado 4.3: Sintáxis del comando para crear una UDF en MySQL

dicción. Usamos la misma función de predicción generada por el *back-end* de C. Al final, se invoca la función `deinit( )`. Esta función se utiliza para liberar cualquier recurso.

El Listado 4.4 muestra un código de ejemplo de las funciones prototipo presentadas en la Figura 4.3. Es importante incluir el encabezado de la biblioteca `"mysql.h"`. Con estas funciones, podemos implementar una UDF en MySQL de la siguiente manera. El código fuente debe compilarse como un archivo de biblioteca compartida. Después de compilar una biblioteca compartido que contenga la UDF, debe instalarse y copiarse en el directorio de MySQL. Luego, dentro de MySQL se debe emitir un comando para crear una función. En en Listado 4.3 se muestra la sintaxis del comando para la creación de funciones definidas por el usuario.

El *function_name* es el nombre que debe usarse en las declaraciones SQL para invocar la función. La cláusula de `RETURNS` depende del tipo de datos devueltos por la UDF. *shared_library_name* es el nombre base de la biblioteca compartida creada que contiene el código que implementa la UDF.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <math.h>
#include "mysql.h"

my_bool myudf_init( UDF_INIT* initid, UDF_ARGS* args, char* message){
  if (args->arg_count != N){
      strcpy(message,"Wrong number of arguments: It requires N arguments");
    return 1;
  }
  args->arg_type[0] = REAL_RESULT;
  args->arg_type[1] = REAL_RESULT;
  ...
  args->arg_type[N] = REAL_RESULT;
  return 0;
}

char * myudf(UDF_INIT *initid, UDF_ARGS *args __attribute__((unused)), char *result,
     unsigned long *length, char *is_null, char *error __attribute__((unused))){
  double A=(double) *((double*)args->args[0]);
  double B=(double) *((double*)args->args[1]);
  ...
  double N=(double) *((double*)args->args[N]);
  sprintf(result, "%s", classification(A, B, ..., N));
  *is_null= 0;
  *length= (uint) strlen(result);
  return result;
}

char const * classification(double A, double B, double ..., double N){
  ...
  return prediction;
}

void myudf_deinit(UDF_INIT *initid){
  ...
}
```

Listado 4.4: Exemplo de los prototipode código generado para ser desplegado en MySQL para crear una función definida por el usuario.

Vertica. Esta es una base de datos analítica orientada a columnas que puede ejecutarse en un clúster de servidores. Vertica permite construir UDFs utilizando el lenguaje C ++ o Java. Optamos por usar C ++ para generar UDFs. Utilizamos plantillas para los *wrappers* de funciones definidos por Vertica en C++ y utilizamos el *back-end* de C para generar las funciones que implementan la función de predicción de un modelo predictivo. Para construir el código de una UDF en Vertica, se debe agregar el encabezado de la biblioteca `"vertica.h"` y se deben crear dos clases. La primera clase debe heredar de las clases definidas de Vertica según el tipo de UDF. Por ejemplo, Vertica tiene funciones escalares, funciones agregadas, funciones de filtro y funciones analíticas. La primera clase contiene

```
1 virtual ScalarFunction *createScalarFunction( )
2 virtual AggregateFunction *createAggregateFunction( )
```

Listado 4.5: Prototipo de las funciones para indicar el tipo de UDF a ser creada en Vertica

una función `processBlock( )`. Esta función se encarga de procesar un bloque de filas. Dentro de la función `processBlock( )` se validan los argumentos de la función y se define un bucle principal. Este ciclo itera sobre cada fila, dentro del ciclo invocamos la función de predicción y almacenamos el resultado en una variable. La segunda clase debe heredar de un conjunto de clases *Factory* dependiendo del tipo de UDF. Dentro de la segunda clase se define una función llamada `getPrototype`. Esta función se encarga de definir el número de argumentos, su tipo y el tipo de devolución de la UDF. Se debe definir otra función llamada `getReturnType( )`. Dentro de esta función debemos proporcionar el tipo, la longitud, la escala o la información de precisión del valor de retorno de la UDF. Se debe definir otra función para crear una instancia del tipo de UDF. Por ejemplo, si la UDF es escalar entonces el prototipo de la función es como se muestra en la línea 1 de Listado 4.5, si la UDF es de agregación entonces el prototipo de la función es como la linea 2 del Listado 4.5.

Para implementar una UDF en Vertica, el código fuente debe compilarse en una biblioteca dinámica compartida utilizando el SDK de Vertica. Luego, se deben ejecutar los comandos que se muestran en el Listado 4.6 para poder crear la UDF dentro de Vertica. Primero, se debe crear una biblioteca de Vertica indicando la biblioteca compartida que ya se compiló con el compilador de C++. Luego, se crea una función indicando el nombre de la función a ser invocada por las consultas SQL. Debemos especificar el idioma del código UDF y la biblioteca que creamos dentro de Vertica. Finalmente, la función importada se puede invocar a través de una consulta SQL. Un ejemplo de un prototipo UDF para una función escalar se muestra en el Listado 4.7. La función de predicción está contenida en la primera clase y se invoca dentro del bucle principal de la función `processBlock`. En

```
1 CREATE LIBRARY shared_library_name AS file_path;
2 CREATE FUNCTION function_name
3 AS LANGUAGE 'C++'
4 NAME 'function_nameFactory'
5 LIBRARY shared_library_name;
```

Listado 4.6: Comandos para crear una UDF en Vertica

este ejemplo, todos los argumentos son de tipo *Float*, y el retorno es un *varchar* (una cadena). Después de invocar el UDF en una declaración SQL, los resultados se proyectan en una tabla.

4.3.4 Multi-Core C con OpenMP

Al extender el *back-end* desarrollado para el lenguaje C con OpenMP podemos aprovechar las arquitecturas de cómputo multi-core. Con OpenMP podemos expresar paralelismo de memoria compartida usando la misma estructura del código secuencial [93]. En nuestro código fuente generado en el lenguaje C, agregamos el encabezado "omp.h ". Las construcciones OpenMP se pueden agregar al código que se puede paralelizar. Por ejemplo, en las funciones *kernel* de los modelos de máquina de vectores de soporte podemos aplicar una reducción dentro de un *for loop* que itera sobre cada vector de soporte. Una reducción es una primitiva utilizada en el contexto de la programación paralela para combinar múltiples vectores en uno, utilizando un operador binario asociativo. En el ejemplo del Listado 4.8, indicamos que *for loop* se ejecuta en paralelo en múltiples procesadores agregando un comando OpenMP. En este comando indicamos que estamos reduciendo el *for loop* y estamos usando la variable `R` como acumulador.

Para habilitar OpenMP, requerimos agregar la bandera OpenMP `-fopenmp` durante la compilación, de lo contrario nuestras directivas y construcciones serán ignoradas. Dado que definimos plantillas para los diferentes tipos de modelos predictivos, hemos identificado en qué partes podemos agregar estos tipos de construcciones para paralelizar la ejecución del código. En las plantillas ya sabemos dónde se puede paralelizar un ciclo. Por ejemplo, un

```
#include "Vertica.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <math.h>

class exception;
using namespace Vertica;
class concreteprediction_class : public ScalarFunction{

  virtual void processBlock(ServerInterface &srvInterface,BlockReader &argReader,
    BlockWriter &resWriter){
    try {
      if (argReader.getNumCols() != N)
        vt_report_error(0, "Function only accept N arguments, but %zu provided",
    argReader.getNumCols());
      do {
        const vfloat A = argReader.getFloatRef(0);
        const vfloat B = argReader.getFloatRef(1);
        ...
        const vfloat N = argReader.getFloatRef(N);

        resWriter.getStringRef().copy(classification(A, B, ..., N));
        resWriter.next();
      } while (argReader.next());
    } catch(std::exception& e) {
      vt_report_error(0, "Exception while processing block: [%s]", e.what());
    }
  }

  char const * classification(double A, double B, double ..., double N){
    ...
    return prediction;
  }

};

class concrete_classFactory : public ScalarFunctionFactory {
  virtual ScalarFunction *createScalarFunction(ServerInterface &interface){
    return vt_createFuncObject<predicted_default_nnet_class>(interface.allocator);
  }
  virtual void getPrototype(ServerInterface &interface, ColumnTypes &argTypes,
    ColumnTypes &returnType){
    argTypes.addFloat();
    argTypes.addFloat();
    ...
    argTypes.addFloat();
    returnType.addVarchar();
  }
  void getReturnType(ServerInterface &srvInterface, const SizedColumnTypes &argTypes
    , SizedColumnTypes &returnType) {
    returnType.addVarchar(255);
  }
};

RegisterFactory(concrete_classFactory);
```

Listado 4.7: Ejemplo de los prototipos del código generada para ser desplegado en Vertica como una UDF.

kernel de una SVM contiene un *for loop* para una reducción, por lo tanto, ya sabemos que podemos agregar una construcción OpenMP. Otro ejemplo es el caso de las redes neuro-

```
1 #pragma omp parallel for reduction(+:R)
2 for(int i=0;i < n; i++)
3   R +=some_computation
```

Listado 4.8: Ejemplo de implementar una reducción paralela con OpenMP en un ciclo for

nales, en los cómputo de la red (*feed-forward*, las neuronas de la misma capa se pueden calcular de forma independiente. Por lo tanto, el proceso puede dividirse entre múltiples núcleos y lograr un mejor rendimiento en términos de tiempos de ejecución.

4.3.5 CUDA-C para GPUs

Las unidades de procesamiento gráfico se diseñaron originalmente para acelerar la creación de imágenes y están destinadas a generar la salida en un dispositivo de visualización. Sin embargo, su estructura altamente paralela los hace muy eficientes para calcular expresiones de álgebra lineal. Debido a esto, se han utilizado ampliamente para el aprendizaje automático. Para aprovechar las unidades de procesamiento gráfico, diseñamos una estructura general para generar código fuente que realiza el cómputo de modelos predictivos utilizando una GPU. Para implementar esto, optamos por generar código CUDA C. La plataforma CUDA permite expresar paralelismo a través de una extensión del lenguaje C++.

Cuando se desarrolla una aplicación diseñada para ejecutarse en una GPU, se deben definir al menos dos tipos de funciones. Una función maestro y una esclavo (conocido como CUDA Kernels). Las funciones tipo esclavo se ejecutan en la GPU utilizando hilos. Estos hilos se distribuyen entre los núcleos CUDA de la GPU. Cada hilo ejecuta las instrucciones definidas en la función esclavo y varios hilos se ejecutan en paralelo. La ejecución en GPU sigue el enfoque de Datos múltiples de instrucción única (SIMD), donde un conjunto de valores de datos se procesa en paralelo. La función maestra se encarga de asignar y liberar la memoria principal y la memoria del dispositivo (GPU). También copia datos desde y hacia el dispositivo e invoca las funciones esclavas [94].

En el entorno CUDA, los conceptos de *grid* y *block* se utilizan para especificar la canti-

```
1 /*three dimension */
2 dim3 dimBlock(32, 32, 1);
3 dim3 dimGrid(32, 32, 1);
4 /*one dimension */
5 int dimBlock = 1024;
6 int dimGrid = 1024;
7 function_name<<<dimGrid, dimBlock>>>(parameters...)
```

Listado 4.9: Ejemplo de la definición del dimGrid y dimBlock y el prototipo para invocar una función esclavo CUDA

dad de hilos lanzados para calcular la misma función esclava. Un *block* es un grupo de hilos que se ejecutarán en paralelo. Los hilos en el mismo *block* pueden comunicarse a través de la memoria compartida. Una *grid* es un grupo de *blocks*. En una *grid*, cada *block* tiene el mismo número de hilos. Cada *block* es un arreglo de hilos de una, dos o tres dimensiones definida por las dimensiones D_x, D_y y D_z. Cada hilo tiene un índice de hilo (x, y, z). Una *grid* puede ser una arreglo de *blocks* de una, dos o tres dimensiones. El número de hilos en un *block* generalmente depende del número de hilos necesarios para resolver un problema en particular. La cantidad total de hilos lanzados al invocar una función esclava se calcula mediante dos parámetros; dimGrid y dimBlock. Estos parámetros pueden ser de una, dos o tres dimensiones. La cantidad total de hilos se obtiene multiplicando el tamaño de la *grid* (número de *blocks*) y el tamaño de los *blocks* (número de hilos). El código presentado en el Listado 4.9 son ejemplos válidos de definiciones dimGrid y dimBlock, y el prototipo para invocar una función CUDA.

Una de las principales ventajas que ofrecen las GPU es que los cálculos se pueden distribuir entre múltiples núcleos. Por lo tanto, un enfoque directo es distribuir un lote de observaciones entre dichos múltiples núcleos para que cada núcleo genere una predicción. El cuello de botella principal para usar una GPU es la copia de datos entre la memoria principal y la memoria del dispositivo. Por lo tanto, optamos por un diseño que minimizara las transferencias de datos.

El Algoritmo 5 muestra una función maestro genérica que organiza los datos de entrada en lotes e invoca las funciones esclavo para generar predicciones. Esta función maestra re-

cibe una matriz $\vec{M}$. Cada fila en esta matriz es una observación. Las columnas representan las variables de una observación. Para cada observación, se generará una predicción y se almacenará en el vector $\vec{O}$, éste vector es el valor de retorno de la función maestro. Los primeros pasos del algoritmo maestro de la línea 1 a la línea 3 es asignar memoria en el dispositivo y copiar variables específicas del modelo predictivo de la memoria principal al dispositivo. De la línea 4 a la 10, calculamos un tamaño de lote que quepa en la memoria libre del dispositivo. Según el tamaño del lote, podemos calcular el número total de lotes en que se va a particionar el conjunto de datos y el número de observaciones por lote. Aunque en el Algoritmo 5 el tamaño del *block* es de 1024, esto es solo un ejemplo. El tamaño de *block* se puede modificar y, en función de dicho número, podemos calcular el tamaño del *grid*. En las líneas 11 y 12, declaramos los vectores $\vec{R}$ y $\vec{M}'$ de longitud igual al número de observaciones por lote. En $\vec{R}$ almacenaremos los resultados de las funciones esclavo y en $\vec{M}'$ contendremos un lote de observaciones de $\vec{M}$. En la línea 14, declaramos un bucle principal que itera sobre el número de lotes necesarios para procesar $\vec{M}$. Dentro del bucle principal, usando un *offset*, asignamos un subconjunto de observaciones correspondientes de $\vec{M}$ a $\vec{M}'$, y copiamos $\vec{M}'$ de la memoria principal al dispositivo. Luego, invocamos la función esclavo para calcular las predicciones. La función esclavo almacena los resultados en $\vec{R}$. Desde allí, copiamos $\vec{R}$ del dispositivo a la memoria principal. Se declara un bucle interno donde iteramos sobre cada valor de $\vec{R}$. Dentro del bucle interno, se pueden realizar operaciones adicionales si es necesario y los resultados almacenados en $\vec{O}$. Fuera del bucle principal, en la línea 25, podemos liberar la memoria asignada a las variables del dispositivo. Finalmente, la función maestro devuelve las predicciones en el vector $\vec{O}$.

La función esclavo recibe un bloque de memoria que contiene un lote de observaciones organizadas en una matriz, así como los datos específicos del modelo y el tamaño del lote. Un ejemplo del prototipo para la función esclavo se presenta en Algoritmo 6. Cuando se inicia la función esclavo, cada hilo procesa una fila de la matriz de entrada. Como hemos indicado antes, cada fila corresponde a un registro de datos (observación) que necesita

generar una predicción. En el algoritmo esclavo, validamos que el identificador de hilo es más pequeño que el número de observaciones a procesar. Esta es una validación necesaria en caso de que lancemos más hilos de los necesarios al cumplir con las especificaciones de CUDA ya que es recomendable que la cantidad de hilos a lanzar debe ser potencia de dos. Entre las líneas 3 y 4, se definen operaciones específicas de acuerdo con un modelo predictivo. Al final, los resultados se almacenan en un vector que se copiará nuevamente a la memoria principal. Vale la pena mencionar que las funciones esclavo de CUDA no permiten acceder a las matrices pasadas por parámetros a través de un índice bidimensional. Por lo tanto, las matrices se manejan como vectores de orden de fila mayor (*row-major order*) y se accede a los valores a través de un índice unidimensional.

Para los modelos de máquina de vectores de soporte, la estructura de los Algoritmos 5 y 6 acelera los cálculos de la función de predicción como mostraremos en la Sección 5.3. Se pueden realizar más modificaciones a dichas funciones para lograr un mayor rendimiento, especialmente para los modelos SVM de multi-clase. Dado que las SVM multiclase requieren mucho poder de cómputo, hemos definido diferentes algoritmos para la generación de código. El algoritmo utilizado para la predicción se define por la naturaleza de los vectores de soporte. Una máquina de vectores de soporte contiene los llamados vectores de soporte, que es un grupo de vectores n-dimensionales que se pueden representar como una matriz. Cada fila de la matriz es un vector de soporte, cada columna corresponde a una variable de los datos utilizados en la tarea de capacitación. El número de filas es igual al número de vectores de soporte en un modelo dado. Los vectores de soporte pueden ser una matriz dispersa o densa. Por lo tanto, tenemos las siguientes ramas separadas de generadores de código; para clasificación binaria (que se definió previamente), para multiclase donde los vectores de soporte son densos y para multiclase cuando los vectores de soporte son escasos.

Independientemente de si los vectores de soporte son densos o dispersos, la estructura general de la función maestra diseñada para la GPU es la misma. Para el código generado

Algorithm 5: Pseudocódigo de la función maestra para organizar datos en lotes e invocar las funciones esclavas para generar predicciones usando la GPU.

Input: $\vec{M}$ en donde:
$\vec{M} \leftarrow$ representa los datos de entrada en una matriz de $r \times c$, cada fila representa una observación y una predicción va a ser generada por cada fila.

Output: $\vec{O}$

`/* Declarar constantes y variables del modelo predictivo */`

1 ...
2 Asignar memoria en el dispositivo para las variables declaradas con `cudaMalloc()`
3 Copiar las variables del modelo al dispositivo con `cudaMemcpy()`
4 $free \leftarrow$ espacio libre de la memoria del dispositivo
5 Calcular el tamaño del lote de observaciones que sea menor al espacio libre del dispositivo ($free$) y la cantidad de lotes necesarios para generar las predicciones de todas las observaciones de $\vec{M}$.
$ex \leftarrow$ cantidad de lotes (cantidad de ejecuciones del ciclo principal)
$l \leftarrow$ observaciones a procesar por lote
6 $b \leftarrow 1024$ `/* tamaño fijo del block */`
7 $g \leftarrow l/b$ `/* tamaño del grid */`
8 **if** $(l \ mod \ g) \neq 0$ **then**
9 $\quad g \leftarrow g + 1$
10 **end**
11 $\vec{R} \leftarrow$ un vector para almacenar los resultados de la predicción de manera temporal
12 $\vec{M'} \leftarrow$ un vector para almacenar l observaciones a procesar
13 Asignar memoria en el dispositivo para las variables $\vec{R}$ y $\vec{M'}$ con `cudaMalloc()`
14 **for** $i \leftarrow 0$ ***to*** ex **do**
15 $\quad offset \leftarrow i \times l$
16 $\quad \vec{M'} \leftarrow$ asignar l observaciones (renglones) correspondientes al lote $_i$ of $\vec{M}$ en *row-major order*
17 $\quad$ Copiar $\vec{M'}$ desde la memoria principal a la memoria del dispositivo con `cudaMemcpy()`
18 $\quad$ getPrediction$<<<g, b>>>(M', \ldots, l)$ `/* invocar la función de predicción */`
19 $\quad$ Copiar $\vec{R}$ desde la memoria del dispositivo a la memoria principal con `cudaMemcpy()`
20 $\quad$ **for** $k \leftarrow 1$ ***to*** l **do**
$\quad\quad$ `/* en caso de ser necesario, transformar el resultado para obtener la predicción final */`
21 $\quad\quad$ ...
22 $\quad\quad O_{k+offset} \leftarrow R_k$
23 $\quad$ **end**
24 **end**
25 Liberar la memoria asignada al dispositivo con `cudaFree()`
26 **return** $\vec{O}$

Algorithm 6: Pseudocódigo de la función esclava a ejecutar por la GPU, cada hilo genera una predicción para una observación.

Input: $\vec{X}$, ..., $\vec{T}$, y l, en donde:
$\vec{X} \leftarrow$ un bloque de memoria que contiene un lote de observaciones orgainzados en un vector en *row-major order*
`.../* variables específicas del modelo */`
$\vec{T} \leftarrow$ vector para almacenar los resultados
$l \leftarrow$ un valor escalar indicando la cantidad de observaciones a procesar
Output: Los resultados son almacenados en $\vec{T}$

```
1 tid ← Identificador de hilo
2 if tid < l then
3 |   idx ← tid × n                                  /* Índice de X⃗ */
  |   /* calcular una predicción para la observación X_idx      */
4 |   T_tid ← R
5 end
```

cuando la SVM se representa como una matriz densa, se definen dos funciones esclavo; uno para calcular productos punto y operaciones específicas del núcleo entre los datos de entrada y los vectores de soporte, y una segunda función para calcular la puntuación de cada modelo SVM. El código generado para calcular los productos de puntos se basa en la multiplicación matriz-matriz al cargar submatrices con memoria compartida por *warp* propuesta por Volkov y Demmel en [95]. Un *warp* es la unidad de paralelismo ejecutable más pequeña en un dispositivo CUDA y comúnmente comprende subprocesos de 32.

En el Algoritmo 7 se presenta un pseudocódigo de la función esclavo para la clasificación múltiple de una máquina de vectores de soporte con una matriz densa. Las observaciones están representadas por la matriz $\vec{A}$, los vectores de soporte por la matriz $\vec{B}$, y los resultados se almacenan en la matriz $\vec{C}$. Dado que el diseño original del algoritmo de multiplicación matriz-matriz es para procesar datos en lotes, el algoritmo funciona para al menos lotes de tamaño igual al tamaño del *warp*. La definición de cómo se invoca la función esclava es la siguiente. Suponiendo que el tamaño de *warp* (W) es 32, se lanzan hilos de 1024 por cada observación de 32 utilizando un dimBlock de dos dimensiones de 32×32 ($W \times W$). DimGrid también tiene dos dimensiones. La primera dimensión se define por el número de vectores de soporte dividido por el tamaño de *warp*. La segunda dimensión

se define por el número de observaciones a procesar dividido por el tamaño de la *warp*. En la función esclavo presentada en el Algoritmo 7, se declara un ciclo principal en la línea 10. Este ciclo principal itera sobre los elementos W (tamaño de *warp*) de las matrices A y B. Dentro del ciclo principal, dos Se realizan los pasos principales. Primero, de la línea 11 a la línea 13, cada subproceso carga un valor de las matrices $\vec{A}$ y $\vec{B}$ en dos submatrices de memoria compartida $\vec{AS}$ y $\vec{BS}$ de dimensión $W \times W$. Luego, los hilos se sincronizan. Segundo, de la línea 14 a la línea 17 cada hilo calcula un producto punto entre una fila de matriz $\vec{AS}$ y una columna de matriz $\vec{BS}$ y el resultado se acumula en una variable. Luego, los hilos se sincronizan. Fuera del ciclo principal, se realizan operaciones específicas según el tipo de núcleo. Finalmente, en la línea 19 los resultados se almacenan en el vector $\vec{C}$. Después de que se calculan los productos de punto y las operaciones específicas del núcleo, se invoca una segunda función esclavo para calcular la predicción de cada modelo SVM. En la función maestra, los resultados se vuelven a copiar en la memoria principal. Para cada observación a procesar, se cuentan los votos de las clases ganadoras y la predicción final se almacena en el vector de salida.

En algunos casos, especialmente con modelos cuyos vectores de soporte son muy grandes, muchos coeficientes pueden tener un valor igual a cero. En estos casos, los vectores de soporte pueden representarse mediante una matriz dispersa. Entonces, el código generado es diferente. Los vectores de soporte se representan en un formato de fila dispersa comprimida (CSR). El formato CSR almacena una matriz dispersa M de dimensiones $r \times c$ en forma de filas utilizando tres vectores; $\vec{A}$, $\vec{AC}$ y $\vec{AI}$. Donde $\vec{A}$ contiene todos los valores distintos de cero de M en orden de fila mayor, $\vec{AC}$ tiene la misma longitud que $\vec{A}$ e indica el índice de la columna de un elemento dado de $\vec{A}$. El vector $\vec{AI}$ es el índice de fila y se define de la siguiente manera, $AI_0 \leftarrow 0$ y AI_i es igual a la suma de AI_{i-1} y el número de cero elementos en la fila $i-1$-iésima de la matriz original M. Un ejemplo de una matriz dispersa se presenta en las siguientes expresiones, donde la matriz M se puede representar en un formato CSR por los tres vectores A, AC y AI.

Algorithm 7: Pseudocódigo de la función esclava a ejecutar por la GPU cuando hay un modelo de máquina de vectores de soporte de clasificación múltiple y los vectores de soporte están representados por una matriz densa.

Input: $\vec{A}$, $\vec{B}$, $\vec{C}$, . . ., y l,en donde:
$\vec{A} \leftarrow$ un bloque de memoria que contiene un lote de la matriz de datos organizado en un vector *row-major order*
$\vec{B} \leftarrow$ un bloque de memoria que contiene los vectores de soporte organizados en un vector *row-major order*
$\vec{C} \leftarrow$ un bloque de memoria que contiene una matriz para almacenear los resultados del producto punto y resto de operaciones kernel organizado en un vector *row-major order*
`.../* variables específicas del kernel */`
$l \leftarrow$ un valor escalar indicando la cantidad de observaciones a procesar

Output: Los resultados son almacenados en $\vec{C}$

$b_x \leftarrow$ identificador de bloque en la dimensión x
$b_y \leftarrow$ identificador de bloque en la dimensión y
$t_x \leftarrow$ identificador de hilo de la dimensión x
$t_y \leftarrow$ identificador de hilo de la dimensión y
$row \leftarrow$ renglón actual
$col \leftarrow$ columna actual
$d \leftarrow$ índice del valor de salida
$W \leftarrow 32$ `/* tamaño del warp */`
$c \leftarrow 0$ `/* variable temporal para almacenar el resultado */`
`/* ciclo sobre las sub-matrices hasta que todos los vectores de soporte son procesados */`
for $i \leftarrow 0$ ***to*** l/W **do**
  $AS_{t_y}^{t_x} \leftarrow A_{row}$
  $BS_{t_y}^{t_x} \leftarrow B_{col}$
  Sincronizar hilos
  for $k \leftarrow 0$ ***to*** W **do**
    $c \leftarrow c + AS_k^{t_y} \times BS_{t_y}^k$
  end
  Sincronizar hilos
end
`/* realizar operaciones específicas del kernel */`
$C_d \leftarrow c$

$$M = \begin{Bmatrix} 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 5 \\ 1 & 0 & 0 & 0 \\ 1 & 9 & 0 & 0 \end{Bmatrix} \quad \rightarrow \quad \begin{aligned} A &= [3, 4, 5, 1, 1, 9] \\ AC &= [2, 1, 4, 0, 0, 1] \\ AI &= [0, 1, 3, 4, 6] \end{aligned}$$

Si los vectores de soporte están representados por una matriz dispersa, entonces se puede reducir el número de cálculos necesarios para generar una predicción. Supongamos que tenemos un modelo de máquina de vectores de soporte con los vectores de soporte representados por la matriz M con filas R y columnas C. Se debe realizar un producto punto entre una observación y cada vector de soporte. Esto significa que se requieren $R \times C$ multiplicaciones y sumas. Sin embargo, los coeficientes cero en los vectores de soporte no contribuyen al resultado acumulado. Denotemos Z como el número de coeficientes cero presentes en la matriz M. Luego, al tener la matriz dispersa en un formato comprimido donde todos los valores son diferentes a cero, el número total de multiplicaciones realizadas es $(R \times C) - Z$. Por lo tanto, la cantidad de operaciones se reduce en términos de Z y, como consecuencia, se reduce el costo computacional.

En el Algoritmo 8 se presenta un pseudocódigo para implementar la función esclavo para la máquina de vectores de soporte de multi-clasificación con una matriz dispersa. La función maestro invoca esta función esclavo de la siguiente manera. DimBlock se define como un valor constante. DimGrid se calcula multiplicando el número de vectores de soporte con el tamaño de *warp*. Luego, el resultado de esa multiplicación se divide por el tamaño del *block*. El resultado de la división se multiplica por el número de filas a procesar por lote. Las variables dimBlock y dimGrid son de una dimensión.

El Algoritmo 8 recibe como parámetros; la observación de entrada, los vectores de soporte en formato CSR, variables de kernel específicas, un *offset* de entrada y un *offset* de salida. El *offset* de entrada se usa en la línea 18 e indica a los hilos en qué observación

van a trabajar. El *offset* de salida se usa en la línea 31 e indica la posición en el vector $\vec{C}$ donde se va a almacenar la predicción. Para cada *warp* obtenemos dónde comienza y termina el índice de fila de la observación de acuerdo al índice de fila de los vectores de soporte. El ciclo definido en la línea 17 itera desde el inicio de la fila hasta el índice final de la fila donde se realizan multiplicaciones entre los valores de los vectores de soporte y los valores de entrada. Los resultados se acumulan en una variable. Fuera del ciclo en la línea 21, el resultado acumulado se almacena en un vector de memoria compartida y los hilos se sincronizan. Un segundo ciclo se define en la línea 24 para acumular los valores de resultado de un *warp* al acceder al vector de memoria compartida de resultados. Fuera del segundo ciclo en la línea 29, cuando el índice del hilo dentro de un *warp* es igual a cero, se realizan operaciones específicas del SVM Kernel y los valores finales se almacenan en el vector $\vec{C}$. En la función maestro se realizan los mismos pasos realizados en el enfoque de matriz densa. Se invoca una segunda función esclavo para calcular la puntuación de cada modelo SVM. Los resultados copian en la memoria principal. Finalmente, para cada observación, se cuentan los votos de las clases ganadoras y la predicción final se almacena en el vector de salida.

Algorithm 8: Pseudocódigo de la función esclava que debe ejecutar la GPU para una máquina de vectores de soporte de clasificación múltiple representada por una matriz dispersa.

Input: $\vec{I}$, $\vec{S}$, $\vec{SC}$, $\vec{SR}$, ..., IO, y OO, en donde:
$\vec{I} \leftarrow$ un vector que contiene una observación a procesar
$\vec{S} \leftarrow$ un vector que contiene los vectores de soporte diferentes a cero en formato CSR
$\vec{SC} \leftarrow$ un vector con el índice de la columna de $\vec{S}$
$\vec{SI} \leftarrow$ un vector con el índice del renglón $\vec{S}$
`.../* variables específicas de kernel */`
$IO \leftarrow$ un valor escalar para indicar el *offset* de la observación
$OO \leftarrow$ un valor escalar para indicar el *offset* de la predicción a generar
Output: Los resultados se almacenean en el vector $\vec{C}$

1 $tid \leftarrow$ índice global del hilo
2 $ltid \leftarrow$ índice del hilo dentro de un *block*
3 $wid \leftarrow$ índice global del *warp*
4 $lane \leftarrow$ índice del hilo dentro del *warp*
5 $tlane \leftarrow$ canal del hilo dentro de un *block*
6 $wlane \leftarrow$ canal del *warp* dentro de un *block*
7 $\vec{D} \leftarrow$ vector de memoria compartida del tamaño del *block*
8 $\vec{P} \leftarrow$ matrix de memoria compartida
9 $ws \leftarrow 32$ `/* tamaño del warp */`
10 $c \leftarrow 0$ `/* variable temporal para acumular el resultado */`
11 **if** $wid < |S|$ **then**
`/* almacenar el índice del inicio y del final de los vectores del soporte para el warp en una variable de memoria compartida */`
12 **if** $tlane < 2$ **then**
13 $P_{tlane}^{wlane} \leftarrow SR_{wid+tlane}$
14 $rowstart \leftarrow P_0^{wlane}$
15 $rowend \leftarrow P_1^{wlane}$
16 $V_{T_j} \leftarrow V_{T_j} + 1$
17 **for** $i \leftarrow rowstart$ ***to*** $rowend$ **do**
18 $c \leftarrow c + I_{SC_i+IO} \times S_i$
19 $i \leftarrow i + ws$
20 **end**
21 $D_{ltid} \leftarrow c$
22 Sincronizar hilos
23 $tc \leftarrow ws/2$
24 **while** $tc >= 2$ **do**
25 $D_{ltid} \leftarrow D_{ltid} + D_{ltid+tc}$
26 Sincronizar hilos
27 $tc \leftarrow tc/2$
28 **end**
29 **if** $lane == 0$ **then**
30 $D_{ltid} \leftarrow D_{ltid} + D_{ltid+1}$
`/* Compute kernel specific operations */`
31 $C_{OO} \leftarrow D_{ltid}$

CAPÍTULO 5
EVALUACIÓN EXPERIMENTAL

En este capítulo, se presentan y discuten los resultados experimentales de ejecutar código generado por el compilador en diferentes escenarios. La evaluación experimental se utiliza para validar el compilador de múltiples objetivos de dos maneras. La primera validación es la corrección del código generado. Logramos esto mediante el uso de los mismos datos de entrada para el código generado y la herramienta analítica donde se construyeron los modelos predictivos, las predicciones generadas en ambos deben coincidir. Esto es para asegurar que el código generado sea correcto. Esto no se establece explícitamente en cada experimento descrito en este capítulo. Sin embargo, los valores de predicción producidos por el código generado automáticamente por el compilador propuesto siempre se validaron contra los valores de predicción generados por la herramienta de modelado. Una vez que se demostró que los resultados eran correctos, la segunda forma de validación es probando la eficiencia de la ejecución de los modelos predictivos utilizando el código generado por el compilador. Por lo tanto, el tiempo de ejecución para generar predicciones utilizando el código generado por el compilador propuesto se comparó con el tiempo de ejecución de las herramientas de modelado para realizar la misma tarea. En la Sección 5.1 presentamos una descripción de los conjuntos de datos utilizados en las diferentes evaluaciones experimentales. La Sección 5.2 presenta una comparación del tiempo de ejecución entre el código generado para C y Java con la herramienta estadística R. La Sección 5.3 presenta una comparación del tiempo de ejecución del código generado para diferentes arquitecturas de destino usando el lenguaje C; C secuencial, C multinúcleo y CUDA C para GPU. Finalmente, la Sección 5.4 presenta los resultados de los experimentos para ejecutar máquinas de vectores de soporte de clases múltiples en GPU utilizando el código fuente generado y la herramienta de modelado ThunderSVM.

5.1 Conjuntos de datos

Para evaluar el rendimiento del código generado por el compilador multi-destino, se utilizan conjuntos de datos sintéticos y del mundo real. En esta sección, describimos los conjuntos de datos utilizados en la evaluación experimental del compilador multi-destino propuesto.

- **Default of Credit Card Clients (DCCC)**. Este conjunto de datos contiene información de los titulares de tarjetas de crédito de un banco taiwanés. La variable de respuesta - *pago predeterminado* - indica si el titular de la tarjeta ha pagado o no. Este conjunto de datos fue presentado inicialmente por Yeh y Lien en [96]. El tamaño de este conjunto de datos es 2.90 MB. Tiene $30,000$ filas con 24 columnas de variables numéricas y categóricas.

- **Electrical Grid Stability Simulated (EGSS)**. Este es un conjunto de datos simulado de una red eléctrica presentado por Arzamasov et al. en [97]. Incluye solo datos numéricos y tiene $10,000$ filas y 14 columnas. La variable de respuesta indica si los datos de una fila dada representan una red eléctrica estable o inestable.

- **Online Shoppers' Purchasing Intention (OSPI)**. Este conjunto de datos contiene información de compradores en línea de un sitio web. Publicado originalmente por Sakar et al. [98]. Contiene $12,330$ filas, cada fila representa una sesión de comprador diferente durante un período de 1 año. Tiene 18 atributos; ocho categóricos y 10 numéricos. El tamaño de este conjunto de datos es de 1.00 MB. La variable de respuesta indica si una sesión de usuario determinada generó ingresos.

- **MNIST.** Es una base de datos de imágenes de dígitos escritos a mano. Los dígitos han sido normalizados por tamaño y centrados en una imagen de tamaño fijo. Las imágenes tienen una dimensión de 28×28 píxeles. Este conjunto de datos tiene un conjunto de entrenamiento de $60,000$ observaciones, y un conjunto de prueba de

$10,000$ observaciones. Para fines de la evaluación experimental, las imágenes se han transformado en un vector de una dimensión de valores de 784 y se han normalizado. Por lo tanto, los datos utilizados estaban en un formato CSV donde la primera columna es la etiqueta y las siguientes columnas 784 son los valores de píxeles en un orden de fila mayor. Este conjunto de datos se presentó originalmente en [99].

- **RCV1.** Este es el conjunto de datos de Reuters Corpus Volumen 1 [100]. Es un conjunto de datos de categorización de texto completamente etiquetado que consiste en historias de noticias que han sido codificadas manualmente. Para los experimentos, usamos la versión de este conjunto de datos descrita en [101]. En la versión, utilizamos el conjunto de datos original preprocesado eliminando el nivel superior de categorías, con un total de 53 clases. El conjunto de datos de entrenamiento contiene $15,564$ filas. Para las pruebas, obtuvimos al azar $15,564$ filas del conjunto de datos original. El número total de columnas de este conjunto de datos es $47,236$.

- **SensIT.** Este es un conjunto de datos de clasificación de vehículos recopilado durante un experimento realizado en Twentynine Palms, California, en noviembre de 2001. Este conjunto de datos contiene datos de redes inalámbricas de sensores distribuidos, que realizan tareas como detección, clasificación, localización y seguimiento de uno o más objetivos del vehículo. El conjunto de datos tiene tres clases, tiene $78,823$ filas para entrenamiento y $19,705$ filas para pruebas con características de 100. Se utiliza la versión " combinada " del conjunto de datos SensIT Vehicle [102].

- **Connect-4.** Este conjunto de datos contiene todas las posiciones legales de 8 capas en el juego de connect-4 en el que ninguno de los jugadores ha ganado todavía, y en el que el siguiente movimiento no es forzado. x es el primer jugador; o el segundo. La clase de resultado es el valor teórico del juego para el primer jugador. El conjunto de datos utilizado utilizó una codificación binaria para cada característica (o, b, x). Este conjunto de datos está disponible en el repositorio de aprendizaje automático de

UCI [103]. Contiene 3 clases, con $675,577$ filas y 126 características.

- **Poker Hand.** Este es el conjunto de datos de la mano de poker, cada registro es un ejemplo de una mano que consta de cinco cartas jugadas de un mazo estándar de 52. Cada carta se describe usando dos atributos para un total de 10 características. El conjunto de datos de entrenamiento contiene $17,766$ filas, mientras que el conjunto de datos de prueba contiene $1,000,000$ filas. El conjunto de datos se publicó originalmente en [104].

- **Satimage.** Este conjunto de datos contiene valores multiespectrales de píxeles en 3×3 vecindades de imágenes de satélite y la clasificación asociada con el píxel central en cada vecindad. Con este conjunto de datos, el objetivo es predecir la clasificación de las imágenes dados los valores multiespectrales. Este conjunto de datos se publicó originalmente en el repositorio de aprendizaje automático de UCI [103]. Contiene 36 características, seis clases, el conjunto de datos de entrenamiento contiene $4,435$ filas y el conjunto de datos de prueba contiene $2,000$ filas.

5.2 Modelos lineales generalizados usando R y código generado en C y Java

En esta sección, presentamos los resultados de la eficiencia de ejecución del código generado por el compilador para modelos GLM. Presentamos la configuración de los experimentos en tres escenarios diferentes. En el primer escenario, el código generado en el lenguaje C se usa para ejecutar modelos. En un segundo escenario, utilizamos código generado en el lenguaje Java. Para el tercer y último escenario, informamos el tiempo de ejecución de los modelos utilizando la función `predict( )` de R.

Para este conjunto de experimentos utilizamos los conjuntos de datos; DCCC, EGSS y OSPI. Se presentó una descripción detallada en la Sección 5.1. Los conjuntos de datos que utilizamos variaron en la cantidad de registros, tipos de datos y cantidad de variables. Para tener experimentos consistentes, todos los conjuntos de datos originales se guardaron

Datos	Tamaño (MB)	Renglones	Columnas	Tipo de Atributos	Valores a predecir	
					Positiva	**Negativa**
DCCC	2.90	30,000	24	Numérico / Categórico	Default	Non-default
EGSS	2.40	10,000	14	Numérico	Stable grid	Unstable grid
OSPI	1.00	12,330	18	Numérico / Categórico	Revenue True	Revenue False

Tabla 5.1: Resumen de los conjuntos de datos utilizados en la evaluación experimental de GLM's.

como archivos ASCII utilizando valores separados por comas (CSV). La tabla 5.1 muestra un resumen de las características principales de cada conjunto de datos.

5.2.1 Configuración experimental

Los experimentos se ejecutaron en una*workstation* con un Intel Xeon W-2133 CPU con 6 núcleos a 3.60 GHz y 32 GB de RAM. Ubuntu 18.04 se utilizó como sistema operativo y R con la versión 3.4.4. Utilizamos GCC 7.3.0 para compilar el código C y OpenJDK 10.0.2 para compilar el código Java. El flujo de trabajo para configurar los experimentos se dividió en dos pasos principales; modelado y despliegue.

Durante el paso de modelado, para cada conjunto de datos se realizó un breve análisis exploratorio de datos y selección de características. Cada conjunto de datos se dividió en dos subconjuntos; un conjunto de datos de entrenamiento con $75\,\%$ de los datos y un conjunto de datos de prueba con $25\,\%$ de los datos. Con cada conjunto de datos de entrenamiento se construyó un modelo lineal generalizado, para los conjuntos de datos EGSS y OSPI se utilizó la función `glm( )`. Para el conjunto de datos DCCC, el paquete caret [105] se utilizó para realizar un entrenamiento de validación cruzada con su función `train( )`, estableciendo a *glm( )* como método de entrenamiento. Con los conjuntos de datos de prueba, se obtuvo la precisión de cada modelo.

La Tabla 5.2 muestra un resumen de las características principales de los modelos creados para cada conjunto de datos y tres mediciones de rendimiento: *accuracy*, *precision* y

Conjunto de datos	Variables	Tipos de variables	Familia - Función enlace	Accuracy	Precision	Recall
DCCC	7	Numérica	binomial - logit	0.67	0.66	0.69
EGSS	8	Numérica	binomial - logit	0.77	0.70	0.92
OSPI	10	Numérica / Categórica	binomial - logit	0.75	0.68	0.94

Tabla 5.2: Resumen de las características y métricas de performance de los modelos lineales generalizadas construidos con cada conjunto de datos.

Medida	Accuracy	Precision	Recall
Ecuación	$\frac{TP+TN}{TP+FP+FN+TN}$	$\frac{TP}{TP+FP}$	$\frac{TP}{TP+FN}$

Tabla 5.3: Métricas de performance utilizadas en los modelos de los experimentos y su ecuación respectiva.

recall. La Tabla 5.3 contiene la fórmula de cada medición de rendimiento utilizada para *glm's*, donde TP son los positivos verdaderos que se refieren a los valores positivos predichos correctamente. TN se refiere a las observaciones negativas verdaderas, que se predijeron correctamente y fueron valores negativos. FP son las observaciones que se pronosticaron incorrectamente como negativas cuando el valor era positivo. FN son las observaciones que se pronosticaron incorrectamente como positivas cuando el valor era negativo. Los valores objetivo positivos y negativos se definen en términos de los datos. Cuando tenemos dos clases separadas, identificamos una como un valor objetivo positivo y el resto como un valor objetivo negativo. La tabla 5.1 contiene los valores objetivo positivos y negativos de cada conjunto de datos de la configuración experimental.

Después de la construcción del modelo, el compilador multi-destino se utilizó para generar código fuente para los lenguajes C y Java. Se creó una maqueta de software en producción, que lee los datos de los archivos CSV e invoca las funciones de predicción. Regularmente para el software en producción no hay necesidad de esto porque el vector para procesar está disponible en su formato nativo. Realizamos la evaluación experimental de la siguiente manera. Dentro del código de la maqueta, se agregaron instrucciones para medir el tiempo transcurrido total. Esto incluye la lectura de datos, la ejecución del

modelo y el almacenamiento de los resultados en la memoria. Para medir el tiempo transcurrido de cada experimento, utilizamos la función `System.nanoTime( )` en Java y el `clock_gettime( )` en C. En R, se desarrolló un *script* para registrar el tiempo transcurrido de carga de datos, la ejecución de la función `predict( )` y el almacenamiento del resultado en una variable. Se realizaron experimentos adicionales escalando el tamaño de cada conjunto de datos en un factor de 100, 200, 400, 600, 800 y 1000. El objetivo de estos experimentos era observar el comportamiento de la ejecución del modelo a medida que aumenta el tamaño de los datos. Cada experimento se repitió 30 veces, se eliminaron los tiempos de ejecución más lentos y más rápidos y se informaron los promedios del resto de valores. Antes de cada ejecución de la implementación del código C y Java, se borró el caché del sistema. En R, después de cada experimento, todos los objetos del espacio de trabajo se borraron y se llamó al recolector de basura utilizando la función `gc( )`.

5.2.2 Resultados

La evaluación experimental se realizó en los tres conjuntos de datos mencionados anteriormente: DCCC, EGSS y OSPI. Realizamos experimentos para los tres escenarios descritos anteriormente: C, Java y R. La tabla 5.4 muestra los resultados del tiempo promedio en segundos para cada configuración. Estos resultados proporcionan evidencia clara que indica que los modelos implementados en el lenguaje C tenían una clara ventaja sobre Java y R. Los modelos implementados en Java son más lentos que C pero más rápidos que R. Esto es más notorio en los experimentos ampliados.

En cuanto al rendimiento, el comportamiento de los resultados de los conjuntos de datos ampliados es similar. Los resultados de tiempo para procesar modelos lineales generalizados en cada escenario de implementación se escalan linealmente con el tamaño de los datos. La figura 5.1 muestra los resultados usando el conjunto de datos OSPI, donde los experimentos que usan el código fuente C son aproximadamente 15 veces más rápidos que R. Usar Java es seis veces más rápido que usar R. Figura 5.2 muestra los resultados

Conjunto de datos	DCCC			EGSS			OSPI		
Escala	**R**	**Java**	**C**	**R**	**Java**	**C**	**R**	**Java**	**C**
1	0.38	0.12	0.03	0.14	0.11	0.03	0.07	0.08	0.02
100	23.94	3.53	1.15	13.20	1.84	1.14	7.26	1.30	0.50
200	47.92	6.85	2.29	26.17	3.47	2.25	14.42	2.39	0.98
400	96.21	13.47	4.63	52.15	6.73	4.49	29.52	4.56	1.94
600	153.22	20.12	6.99	79.01	10.00	6.69	42.35	6.73	2.89
800	194.85	26.74	9.22	103.93	13.28	8.92	59.98	8.88	3.96
1000	239.45	33.40	11.50	132.24	16.53	11.16	72.97	11.07	4.93

Tabla 5.4: Resultados de la evaluación experimental mostrando el tiempo transcurrido en segundos por cada conjunto de datos, cada escala y cada lenguaje del escenario de la implementación de GLM's.

usando el conjunto de datos DCCC, donde los experimentos que usan el código fuente C son aproximadamente 21 veces más rápido que R, y el uso de Java es siete veces más rápido que R. La Figura 5.3 muestra los resultados usando el EGSS conjunto de datos, donde los experimentos que usan el código fuente C son aproximadamente 12 veces más rápidos que R, y el uso de Java es ocho veces más rápido que R.

En resumen, los modelos desplegados en C o Java tienen tiempos de ejecución más rápidos que los modelos ejecutados directamente en R. Este comportamiento es más evidente cuando el tamaño de los datos aumenta considerablemente. La diferencia entre los resultados parece menos relevante con escalas de tamaño de datos más pequeñas. Como se esperaba, los modelos implementados en C son más rápidos que los modelos implementados en Java y que los modelos ejecutados directamente en R. Esto se debe a que el código fuente de C se compila, generando así programas muy eficientes. Los modelos implementados en el lenguaje Java son más rápidos que R. El código Java se ejecuta en la máquina virtual Java que le permite ejecutarse en todas las plataformas compatibles con Java sin recompilación, agregando una pequeña sobrecarga en los resultados del tiempo de ejecución. El propósito de esta comparación es establecer un punto de referencia para el código fuente generado contra R en términos de rendimiento.

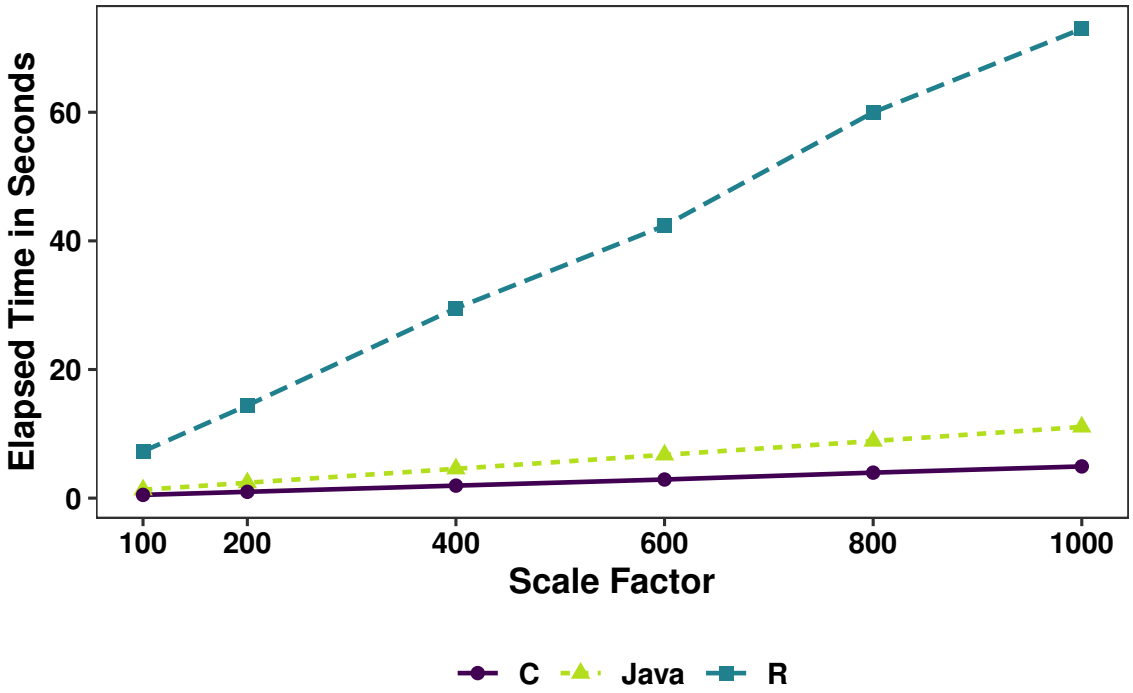


Figura 5.1: Gráfico de líneas con los resultados de la evaluación experimental de un modelo lineal generalizado utilizando la función `predict( )` de R y el código fuente generado en C y Java para el conjunto de datos OSPI con el factor de escala de 100 a 1000.

5.3 Máquina de vectores de soporte binaria usando código generado para CPU de un solo núcleo, CPU de múltiples núcleos y GPU

Esta sección contiene una descripción detallada de una evaluación experimental de la ejecución de modelos de máquinas de vectores de soporte binarias con diferentes versiones del código generado por el compilador de multi-destino. Los modelos SVM pueden potencialmente demandar una gran cantidad de recursos computacionales; por lo tanto, aprovechar las arquitecturas computacionales paralelas, como múltiples núcleos y coprocesadores, siempre es beneficioso. Un paso natural desde la generación de código en lenguaje C usando un enfoque de un solo núcleo es agregar un *back-end* para generar código C para ejecutarse en múltiples núcleos usando el mismo código base. En un paso adicional para lograr un mejor rendimiento, el compilador de multi-destino incluye un *back-end* para generar código CUDA-C.

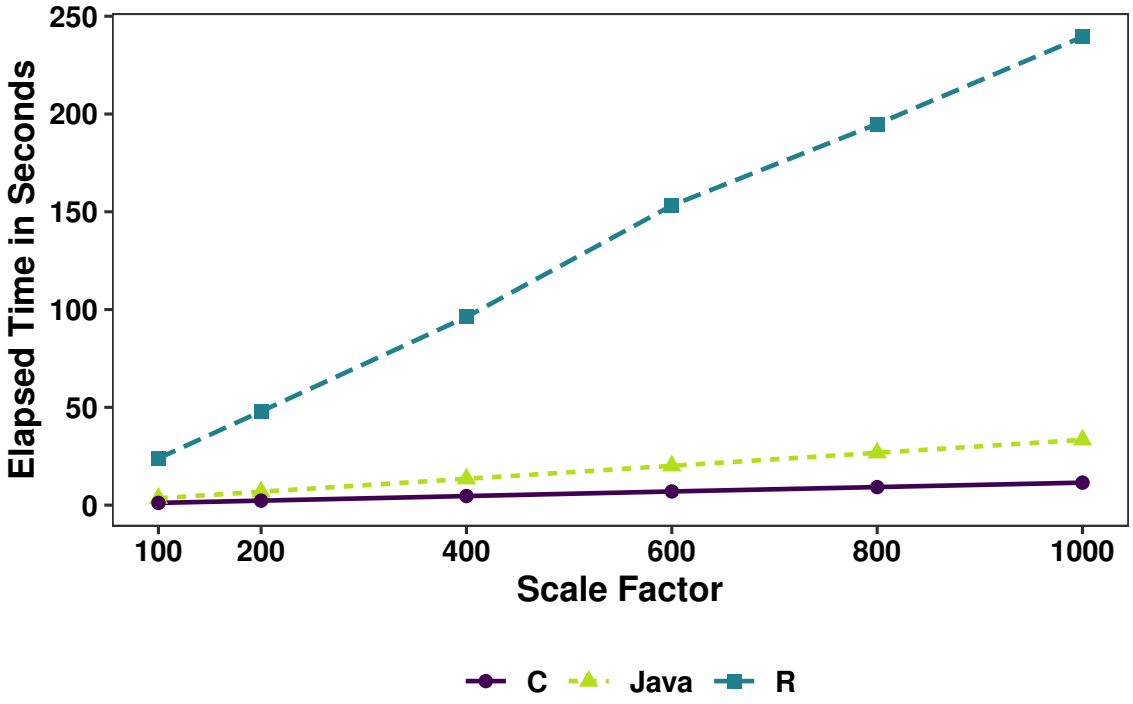


Figura 5.2: Gráfico de líneas con los resultados de la evaluación experimental de un modelo lineal generalizado utilizando la función `predict( )` de R y el código fuente generado en C y Java para el conjunto de datos DCCC con el factor de escala de 100 a 1000 .

En las secciones siguientes presentamos una descripción detallada de los resultados experimentales, el software y el hardware utilizados, los conjuntos de datos y los modelos SVM que construimos. Además, presentamos los resultados de la ejecución de predicción del modelo SVM en tres arquitecturas de implementación diferentes: un núcleo (secuencial), multi-núcleo (multiproceso) y GPU (CUDA). Es importante enfatizar que las evaluaciones de rendimiento presentadas corresponden a tres versiones diferentes de código generadas por el compilador multi-destino. La propuesta de esta disertación se centra en el despliegue de modelos predictivos. Por lo tanto, el objetivo no es construir modelos, sino facilitar la construcción de aplicaciones que integren modelos predictivos como SVM en arquitecturas de CPU y GPU. Utilizamos los *back-end* para la generación de código C para multi-core usando OpenMP y para GPUs usando CUDA. Estos *back-end* se describen en las Secciones 4.3.4 y 4.3.5 respectivamente.

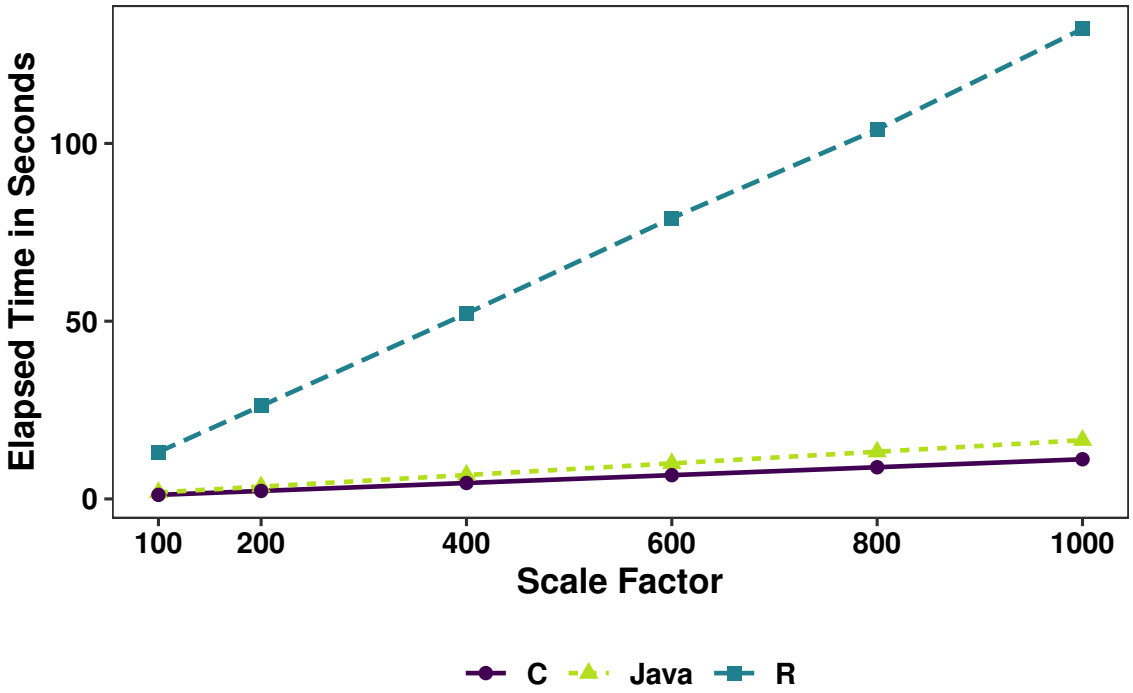


Figura 5.3: Gráfico de líneas con los resultados de la evaluación experimental de un modelo lineal generalizado utilizando la función `predict( )` de R y el código fuente generado en C y Java para el conjunto de datos EGSS con el factor de escala de 100 a 1000.

5.3.1 Configuración experimental

Para los experimentos se utilizó una estación de trabajo con las siguientes características: una CPU Intel Xeon E5-1603 con 4 núcleos a 2.80 GHz y 64 GB de RAM. El sistema operativo utilizado fue Ubuntu 14.04. La GPU utilizada es una GeForce GTX 660. Esta GPU tiene 960 núcleos CUDA en total y 2 GB de RAM. Tiene 5 multiprocesadores de 192 núcleos CUDA y la arquitectura de la GPU es Kepler.

El software estadístico R ([106]) se utilizó para construir un modelo SVM. El modelo se creó utilizando el conjunto de datos DCCC [96] (se presenta una descripción detallada en la Sección 5.1). El conjunto de datos se dividió en un conjunto de entrenamiento con $75\,\%$ de los datos y un conjunto de pruebas con $25\,\%$ de los datos.

Después de realizar un análisis de datos, el modelo SVM fue entrenado con siete variables numéricas. El modelo fue construido usando el paquete `e1071` [107]. Los parámetros

del modelo fueron $C = 1.7$ y $sigma = 0.2$, y el núcleo seleccionado fue lineal. En donde C es el costo de la violación de restricciones y σ es el parámetro de escala de la distribución de Laplace hipotética (media cero) estimada por la máxima verosimilitud. Como medida de performance, se utilizó el AUC (Área bajo la curva), que resultó en 0.72. En resumen, la entrada al modelo es un vector de siete variables numéricas. El modelo contiene $7,509$ vectores de soporte con un núcleo lineal, y el resultado es una etiqueta; " Sí " o " No ", que indica si un cliente va a estar predeterminado o no.

Después del entrenamiento del modelo, el modelo SVM se exportó a un archivo PMML (usando el paquete *pmml* [108]). El archivo PMML es la entrada del compilador de múltiples objetivos, que se utilizó para generar el código fuente necesario para cada entorno donde se implementó; Single-Core, Multi-Core y GPU.

Para la evaluación experimental se utilizó el conjunto de datos de clientes de tarjetas de crédito por defecto. Sin embargo, el conjunto de datos original se incrementó al replicarlo usando varios factores de escala. El objetivo de este experimento es analizar cómo se comportaría cada ejecución de código generado y cómo aumenta el tiempo a medida que aumenta la cantidad de datos a procesar. El conjunto de datos original tiene $30,000$ registros y su tamaño es de alrededor de 2.9 MB. Estos registros se replicaron para crear factores de escala de 10, 20, 40, 80, 100 y 160 con respecto al conjunto de datos original. Para estandarizar la evaluación experimental se construyó una pequeña aplicación. Esta aplicación realiza los siguientes pasos; lee y analiza datos, llama a las funciones de predicción de cada entorno de implementación y finalmente registra el tiempo de ejecución de cada configuración de experimento. Se utiliza el *wall clock* spara registrar el tiempo mediante la función `gettimeofday( )`.

Aunque no se midió el tiempo de I/O (porque el único interés está en el tiempo de ejecución), antes de cada ejecución individual se borró el caché del sistema de archivos. Cada experimento se ejecutó diez veces, eliminando el tiempo más rápido y más lento. Al final, se informó el tiempo promedio de los ocho resultados de cada configuración experimental.

Factor de Escala	Tiempo transcurrido en segundos					
	Precisión doble			Precisión simple		
	Single-core	Multi-core	GPU	Single-core	Multi-core	GPU
1	2.13	1.69	0.50	1.29	0.98	0.48
10	20.83	16.71	0.98	12.68	9.61	0.72
20	41.85	33.69	1.52	25.30	19.19	1.01
40	83.71	67.07	2.63	50.34	38.53	1.63
80	167.32	134.92	4.84	100.83	77.06	2.75
100	210.38	168.10	5.94	126.66	96.91	3.36
160	335.30	269.00	9.29	200.88	156.28	5.17

Tabla 5.5: Resultados del tiempo transcurrido de la ejecución de los experimentos con un modelos de SVM binaria con las diferentes escalas y ambientes de cómputo.

El código se compiló en la *workstation* descrita previamente con banderas de optimización habilitadas; el código para la evaluación también se compiló y se vinculó al código generado. Se realizaron dos versiones de todo el conjunto de experimentos. El primero tenía todos los datos como valores de coma flotante de precisión doble. El segundo tenía todos los datos representados con valores de punto flotante de precisión simple.

5.3.2 Results

La Tabla 5.5 muestra todos los resultados promediados del tiempo transcurrido de la ejecución del modelo SVM con las diferentes escalas y entornos. Se puede ver claramente que los resultados de los tiempos de ejecución de un solo núcleo son más lentos que el resto de los entornos con números de punto flotante de precisión simple y doble. Los resultados de la ejecución multinúcleo con OpenMP son ligeramente más rápidos que los resultados secuenciales. Finalmente, la implementación de GPU con CUDA tiene los tiempos de ejecución más rápidos.

La Tabla 5.6 muestra una perspectiva diferente de los resultados experimentales. Los resultados muestran que, para cada factor de escala, la versión de múltiples núcleos del código es 1.25 veces más rápida que la versión de un solo núcleo cuando se utilizaron

valores de coma flotante de doble precisión. Cuando se usan valores de punto flotante de precisión simple, la versión de múltiples núcleos es 1.30 más rápida que la versión de un solo núcleo. Por otro lado, a medida que aumenta la escala de datos, la velocidad del tiempo de ejecución al usar el código GPU también aumenta considerablemente. Cuando se utilizaron valores de coma flotante de precisión doble, el tiempo de ejecución del código generado para las GPU comienza a ser aproximadamente 4 veces más rápido que el tiempo de ejecución del código secuencial con la escala de datos 1x y termina siendo 36 veces más rápido en los datos de 160x escala. Con precisión simple, la implementación de la GPU comienza a ser 1.67 veces más rápida que la de un solo núcleo en los experimentos de escala de datos 1x, pero termina siendo 38.87 veces más rápida que las ejecuciones de un solo núcleo con la escala 160x.

La ejecución del código generado que usa la GPU es considerablemente más rápida que la ejecución del código generado para la CPU de un solo núcleo. La diferencia entre los tiempos de ejecución aumenta a medida que aumenta el factor de escala de datos. En los experimentos con la escala de datos más grande (160x) con doble precisión, la ejecución del código generado para GPU puede procesar $516,647$ observaciones por segundo, con una ejecución multinúcleo se pueden procesar $17,844$ observaciones por segundo, y con el código generado para ejecución de un solo núcleo $14,316$ observaciones por segundo se pueden procesar. Además, en los experimentos con valores de coma flotante de precisión sencilla, la diferencia es aún mayor. Por ejemplo, utilizando el mismo factor de escala de 160x, la ejecución del código generado para la GPU puede procesar $928,878$ observaciones por segundo, mientras que un solo núcleo puede procesar $23,895$ observaciones por segundo y la ejecución del modelo en múltiples núcleos puede procesar $30,715$ observaciones por segundo . Las Figuras 5.4 y 5.5 muestran una gráfica con los tiempos de ejecución observados en nuestros experimentos. Los tiempos de ejecución de uno y varios núcleos se escalaron linealmente, mientras que los resultados de tiempo de la GPU se escalan sublinealmente a medida que aumenta la escala de datos. El comportamiento en la ejecución

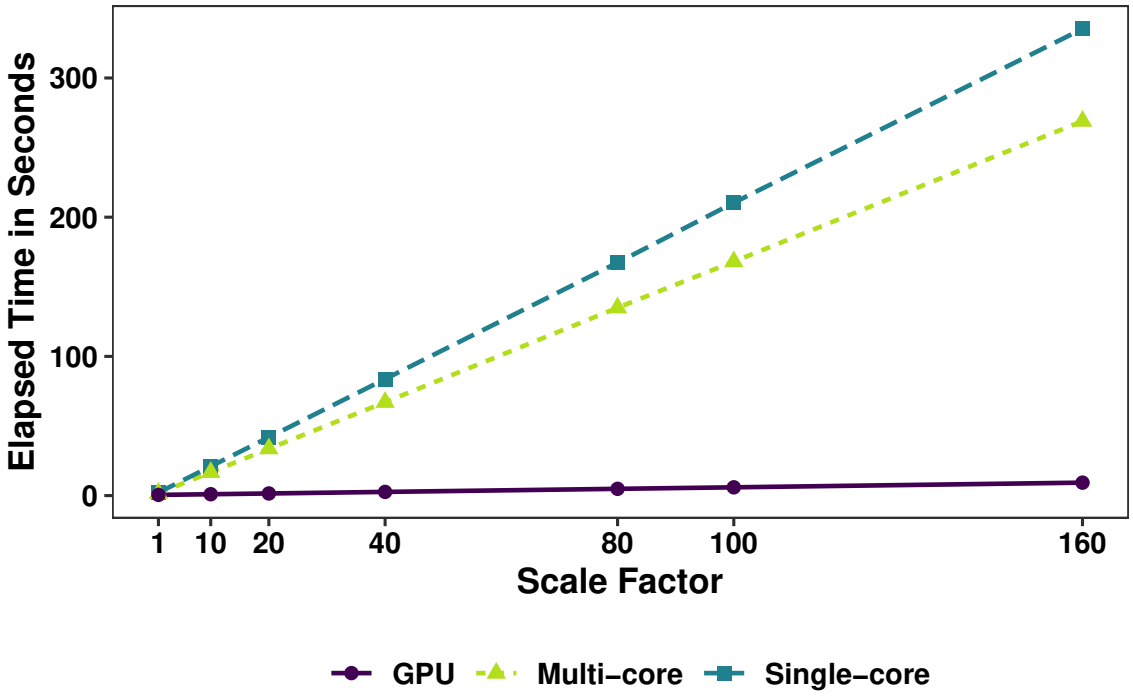


Figura 5.4: Tiempos de ejecución de la ejecución del modelo SVM con las diferentes escalas de datos utilizando valores de coma flotante de doble precisión en los tres entornos: GPU, multi-core y single-core.

de la GPU, especialmente con los factores de escala más pequeños de los datos, se debe al *overhead* relacionado con la transferencia de datos entre la memoria principal y la memoria del dispositivo. Con los factores de escala más bajos (1, 10, 20 y 40) el ancho de banda disponible no se utiliza por completo, mientras que con los factores de escala más altos (80, 100 y 160) el tiempo de ejecución comienza a escalar linealmente porque el ancho de banda disponible del GPU se usa mejor.

5.4 Máquinas de vectores de soporte multi-clase utilizando código generado para GPU y ThunderSVM

En esta sección, presentamos una evaluación experimental de la ejecución de modelos SVM multi-clase en GPUs. La comparación se realiza entre el código generado por el compilador multi-destino y ThunderSVM [109]. ThunderSVM es una herramienta de código

Factor de escala de datos	Ambiente	Precisión doble		Precisión simple	
		Observaciones por segundo	Ganancia	Observaciones por segundo	Ganancia
1	single-core	14,088	1.00	23,208	1.00
	multi-core	17,736	1.26	30,561	1.32
	GPU	59,592	4.23	61,886	1.67
10	single-core	14,301	1.00	23,665	1.00
	multi-core	17,948	1.26	31,223	1.32
	GPU	307,398	21.49	414,770	17.53
20	single-core	14,338	1.00	23,715	1.00
	multi-core	17,811	1.24	31,272	1.32
	GPU	394,218	27.49	593,803	25.04
40	single-core	14,335	1.00	23,838	1.00
	multi-core	17,892	1.25	31,145	1.31
	GPU	455,649	31.79	736,017	30.88
80	single-core	14,344	1.00	23,802	1.00
	multi-core	17,788	1.24	31,144	1.31
	GPU	495,455	34.54	863,817	36.29
100	single-core	14,260	1.00	23,685	1.00
	multi-core	17,847	1.25	30,956	1.31
	GPU	504,742	35.40	894,182	37.75
160	single-core	14,316	1.00	23,895	1.00
	multi-core	17,844	1.25	30,715	1.29
	GPU	516,647	36.09	928,878	38.87

Tabla 5.6: Otra perspectiva de los resultados de los experimentos, presentando las predicciones por segundo generadas en cada ambiente, y la ganancia en términos de velocidad tomando como base la ejecución secuencial.

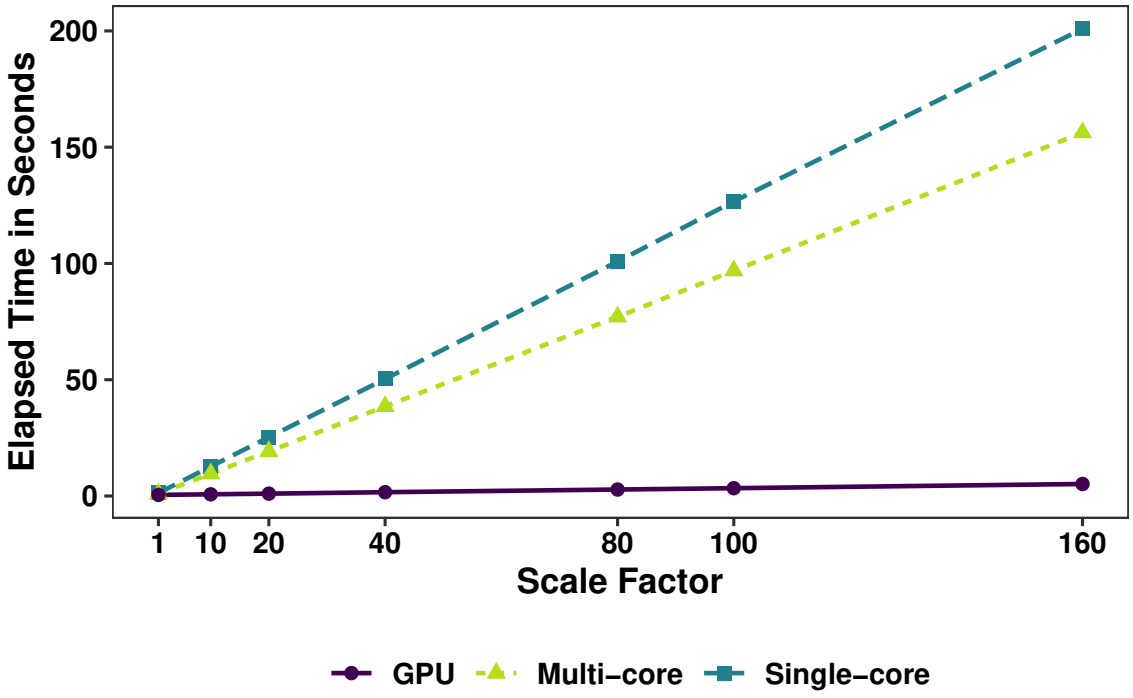


Figura 5.5: Tiempos de ejecución de la ejecución del modelo SVM con las diferentes escalas de datos utilizando valores de coma flotante de simple precisión en los tres entornos: GPU, multi-core y single-core.

abierto para construir y ejecutar modelos de máquinas de vectores de soporte que utilizan GPU y CPU de múltiples núcleos. Esta sección presenta una descripción de los conjuntos de datos utilizados, los modelos construidos y los resultados de los experimentos.

Para estos experimentos utilizamos los siguientes conjuntos de datos: MNIST, RCV1, SensIT, Connect-4, Poker Hand y Satimage. Una descripción detallada de los conjuntos de datos está disponible en la Sección 5.1. La Tabla 5.7 muestra un resumen de las características principales de cada conjunto de datos que utilizamos.

5.4.1 Configuración experimental

Todos los experimentos se ejecutaron en una estación de trabajo con una CPU Intel Xeon W-2133 con 6 núcleos a 3.60 GHz y con 64 GB de RAM. El sistema operativo utilizado fue Ubuntu 18.04. Se utilizó una tarjeta gráfica GeForce GTX 1080. Esta GPU tiene 8

Conjunto de datos	Clases	Variables	Entrenamiento		Pruebas	
			Tamaño (MB)	Observaciones	Tamaño (MB)	Observaciones
MNIST	10	784	154.90	60,000	25.90	10,000
RCV1	53	47,236	1,536.00	15,564	1,536.00	15,564
SensIT	3	100	78.10	78,823	19.50	19,705
Connect-4	3	126	17.20	67,557	N/A	N/A
Poker	10	10	0.58	25,010	23.50	1,000,000
Satimage	6	36	1.40	4,435	0.64	2,000

Tabla 5.7: Resumen de los conjuntos de datos utilizados en la evaluación experimental de la ejecución de máquinas de vectores de soporte multi-clase.

GB de RAM, 2,560 núcleos CUDA y su arquitectura es Pascal. ThunderSVM se compiló para usar valores de coma flotantes de precisión doble en lugar de precisión simple, que es la opción predeterminada. El código generado también utiliza valores de coma flotante de precisión doble, y los datos se tratan también como precisión doble.

Se usaron dos flujos de trabajo para entrenar y desplegar un modelo de máquina de vectores de soporte para cada conjunto de datos. En el primer flujo de trabajo se desarrolló un script, utilizando el lenguaje Python y la biblioteca Scikit-learn, para construir un modelo SVM para cada conjunto de datos. El modelo se exportó al formato PMML utilizando la biblioteca `sklearn2pmml`. El archivo PMML se usó como entrada para el compilador multi-destino y se generó código para implementar los modelos SVM para ejecutar en una GPU. Para usar el código generado, se crearon programas para leer los archivos de datos CSV y para llamar a las funciones para generar predicciones ejecutando el modelo respectivo a partir del código generado por el compilador. En el segundo flujo de trabajo, Python en combinación con la biblioteca ThunderSVM se usó para entrenar un modelo SVM para cada conjunto de datos. Después de que se construyeron los modelos, se persistieron en un formato propietario de ThunderSVM. Para realizar los experimentos, se creó un script de Python, que leelos archivos CSV, carga el modelo guardado previamente en formato ThunderSVM, ejecutar los modelos y finalmente registrar el tiempo transcurrido.

Los conjuntos de datos MNIST y RCV1 tienen muchas características con valores cero,

Conjunto de datos	Parámetros de entrenam.*		Scikit-learn			ThunderSVM		
	Variables	C	Precisión	Vectores de soporte	Tiempo de entrenam. (minutos)	Precisión	Vectores de soporte	Tiempo de entrenam. (minutos)
MNIST	709	10	0.8981	45,625	252.40	0.8982	45,625	2.58
RCV1	20,049	10	0.9558	7,603	210.97	0.9545	7,625	17.98
SensIT	100	10	0.8679	26,179	12.25	0.8676	26,176	0.28
Connect-4	126	10	0.8601	21,491	34.00	0.8603	21,499	0.62
Poker	10	1	0.5857	23,650	13.35	0.5858	26,651	0.92
SatImage	36	10	0.8995	1,204	0.02	0.8995	1,206	0.01

*Nota: Todos los modelos utilizaron el parámetro de entrenamiento $\sigma = 0.125$.

Tabla 5.8: Resumen de los parámetros y resultados de los modelos de máquinas de vectores de soporte utilizando Scikit-learn y ThunderSVM, todos los modelos utilizaron el kernel radial (RBF).

por lo tanto, estas características no son importantes para el entrenamiento del modelo. Filtramos las características no importantes de estos conjuntos de datos. Los modelos SVM resultantes para cada conjunto de datos que utilizan ambos flujos de trabajo son ligeramente diferentes. Esto se debe a que Scikit-learn y ThunderSVM no utilizan el mismo algoritmo de entrenamiento. El algoritmo de entrenamiento Scikit-learn utiliza el CPU, mientras que ThunderSVM usa la GPU. La Tabla 5.8 muestra un resumen de los parámetros para el entrenamiento del modelo y los modelos resultantes para cada conjunto de datos en cada flujo de trabajo. Todos los modelos SVM usan un núcleo radial.

Los modelos SVM generados para los conjuntos de datos MNIST y RCV1 tenían vectores de soporte con muchos coeficientes de valor cero. Por lo tanto, en el código generado, los vectores de soporte se representaron como una matriz dispersa. Los modelos SVM creados con los conjuntos de datos SensIT, Connect4, Poker y Satimage tenían vectores de soporte con muy pocos coeficientes de valor cero. Por lo tanto, los vectores de soporte de esos modelos se representaron con una matriz densa en el código generado.

Se construyeron dos modelos de máquina de vectores de soporte para cada conjunto de datos, uno con Scikit-learn y el otro con ThunderSVM. Para cada modelo, utilizamos

el compilador multi-destino para generar código y luego implementar esos modelos de la siguiente manera. Creamos un código de maqueta para leer datos e invocar la función de predicción de los modelos SVM contenidos en el código generado. Registramos el tiempo la ejecución del código generado por el compilador propuesto usando la función `clock_gettime( )`. Además, también utilizamos los modelos generados con ThunderSVM de la siguiente manera. Creamos un script de Python para cada modelo para leer datos, cargar el SVM previamente entrenado e invocar la función de predicción. Registramos el tiempo de ejecución de los modelos en ThunderSVM usando la función `time.time( )`. Se realizaron experimentos adicionales aumentando la escala de cada conjunto de datos. La intención es observar el comportamiento de la ejecución del modelo a medida que aumenta el tamaño de los datos. Realizamos cada experimento 30 veces y eliminamos los tiempos de ejecución más lentos y más rápidos, informando una media recortada. Los tiempos informados se refieren exclusivamente al tiempo transcurrido de ejecución de la función de predicción de un modelo SVM (es decir, no cronometramos las operaciones de IO).

5.4.2 Resultados

La Tabla 5.9 muestra los resultados de la evaluación experimental de generar predicciones para el conjunto de datos MNIST utilizando los modelos SVM desplegados. En la Tabla 5.9, presentamos el número de registros de datos, el tiempo transcurrido y las observaciones procesadas por segundo para cada factor de escala. El código generado es ligeramente más rápido que ThunderSVM para cada factor de escala, y la ventaja del código generado aumenta con el tamaño de los datos. Para ThunderSVM, los experimentos usando factores de escala 40 y 60 no pudieron ejecutarse porque requerían más de los 64 GB de memoria que tenemos disponible en nuestra estación de trabajo. La Figura 5.6 muestra un gráfico de líneas de los resultados experimentales con el conjunto de datos MNIST hasta un factor de escala de 30.

La Tabla 5.10 muestra el resultado de la evaluación experimental de la ejecución del

Escala	Total de observaciones	Código generado		ThunderSVM	
		Tiempo transcurrido en segundos	Observaciones procesadas por segundo	Tiempo transcurrido en segundos	Observaciones procesadas por segundo
1	60,000	24.97	2,403	28.54	2,102
2	120,000	49.92	2,404	56.36	2,129
4	240,000	106.53	2,253	112.45	2,134
6	360,000	159.78	2,253	168.40	2,138
8	480,000	213.02	2,253	225.04	2,133
10	600,000	266.27	2,253	280.66	2,138
12	720,000	319.55	2,253	336.49	2,140
15	900,000	399.52	2,253	420.51	2,140
20	1,200,000	533.35	2,250	560.66	2,140
30	1,800,000	800.84	2,248	841.15	2,140
40	2,400,000	1,064.91	2,254	N/A	N/A
60	3,600,000	1,600.97	2,248	N/A	N/A

Tabla 5.9: Resultados de los experimentos con modelos SVM multi-clase utilizando la GPU mediante el código generado y ThunderSVM para el conjunto de datos MNIST.

código generado por el compilador que implementa los modelos SVM con el conjunto de datos RCV1. Para este conjunto de experimentos, el código generado es mucho más rápido que ThunderSVM. Usando el código generado, se puede procesar un promedio de $7,781$ observaciones por segundo, mientras que con ThunderSVM se procesa un promedio de solo 183 observaciones por segundo. Estos resultados indican una aceleración de 42x. Para los factores de escala cuatro y ocho que usan el conjunto de datos RCV1, los experimentos no pudieron ejecutarse en ThunderSVM porque no había suficiente memoria en nuestra estación de trabajo para ejecutar el script de Python. La Figura 5.7 muestra un gráfico de líneas de los resultados para los factores de escala uno, dos y tres.

Los resultados de la evaluación experimental con los modelos generados con el conjunto de datos SensIT se presentan en la Tabla 5.11. El código generado es aproximadamente $24\,\%$ más rápido que ThunderSVM. Con el código generado, se pueden procesar en promedio $24,834$ observaciones por segundo, mientras que con ThunderSVM se pueden procesar $19,969$ observaciones por segundo. La Figura 5.8 muestra un gráfico de líneas de los resul-

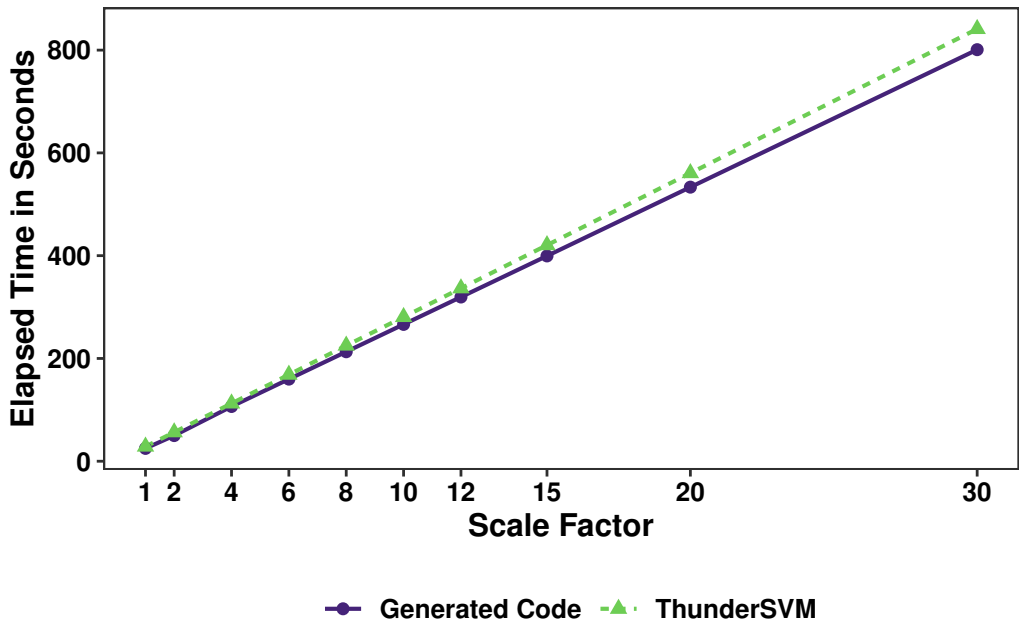


Figura 5.6: Gráfico de líneas de los resultados de la evaluación experimental del código generado y ThunderSVM utilizando la GPU para ejecutar el modelo SVM con los diferentes factores de escala del conjunto de datos MNIST.

tados de todos los experimentos con el conjunto de datos SensIT.

Los resultados obtenidos para los modelos creados con el conjunto de datos Connect-4 se presentan en la Tabla 5.12. El código generado por el compilador es $49\,\%$ más rápido que ThunderSVM. En promedio, se procesan $29,998$ observaciones por segundo ejecutando el código generado, mientras que con ThunderSVM se procesan $20,083$ observaciones por segundo. La Figura 5.9 muestra un gráfico de líneas de los resultados de todos los experimentos con el conjunto de datos Connect-4.

La Tabla 5.13 muestra el resultado de la evaluación experimental de la ejecución de los modelos SVM con el conjunto de datos de Poker. En promedio, con el código generado se pueden procesar $49,033$ observaciones por segundo, mientras que con ThunderSVM se pueden procesar $18,681$ observaciones por segundo. Esto indica una aceleración de 2.62 aproximadamente. La Figura 5.10 muestra un gráfico de líneas de los resultados de todos los experimentos con el conjunto de datos de Poker.

Los resultados de la evaluación experimental usando los modelos SVM creados con el

Escala	Total de observaciones	Código generado		ThunderSVM	
		Tiempo transcurrido en segundos	Observaciones procesadas por segundo	Tiempo transcurrido en segundos	Observaciones procesadas por segundo
1	15,564	1.81	8,617	88.70	175
2	31,128	3.59	8,668	164.07	189
3	46,692	4.69	9,961	248.80	187
4	62,256	6.23	9,992	N/A	N/A
8	124,512	11.67	10,671	N/A	N/A

Tabla 5.10: Resultados de los experimentos con modelos SVM multi-clase utilizando la GPU mediante el código generado y ThunderSVM para el conjunto de datos RCV1.

Escala	Total de observaciones	Código generado		ThunderSVM	
		Tiempo transcurrido en segundos	Observaciones procesadas por segundo	Tiempo transcurrido en segundos	Observaciones procesadas por segundo
1	78,823	3.20	24,667	4.13	19,077
5	394,115	15.85	24,872	19.75	19,951
10	788,230	31.66	24,895	39.15	20,132
15	1,182,345	47.56	24,861	58.99	20,045
20	1,576,460	63.41	24,860	78.55	20,070
25	1,970,575	79.27	24,860	98.27	20,053
30	2,364,690	95.11	24,862	117.37	20,148
40	3,152,920	126.82	24,862	156.48	20,149
80	6,305,840	254.57	24,770	313.69	20,102

Tabla 5.11: Resultados de los experimentos con modelos SVM multi-clase utilizando la GPU mediante el código generado y ThunderSVM para el conjunto de datos SensIT.

conjunto de datos Satimage se presentan en la Tabla 5.14. Los resultados indican una aceleración de 3.62 del código generado con respecto a ThunderSVM. En promedio, se pueden procesar 451,601 observaciones por segundo, mientras que con ThunderSVM 124,586 observaciones por segundo se pueden procesar. Las Figura 5.11 muestra un gráfico de líneas de los resultados de todos los experimentos con el conjunto de datos Satimage.

Según los resultados experimentales, se obtiene una ganancia en términos de eficiencia de ejecución con el código generado por el compilador. Esta ganancia se debe principalmente a que hemos identificado casos especiales de modelos SVM. Los casos especiales se

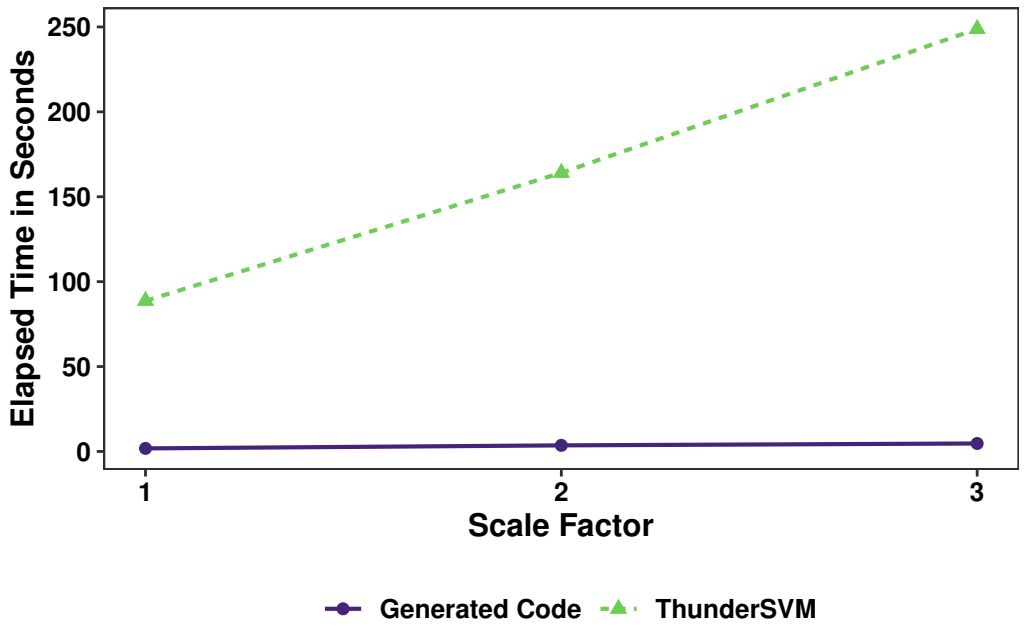


Figura 5.7: Gráfico de líneas de los resultados de la evaluación experimental del código generado y ThunderSVM utilizando la GPU para ejecutar el modelo SVM con los diferentes factores de escala del conjunto de datos RCV1.

pueden desarrollar e integrar en las plantillas de representación intermedias. En el compilador, cuando se identifica un caso especial, podemos usar una plantilla para generar código optimizado. Por ejemplo, los resultados que utilizan los modelos creados para MNIST y RCV1 proporcionan evidencia de que el código generado para calificar un SVM como una matriz dispersa es más rápido que su contraparte. Un caso especial es el conjunto de datos RCV1 donde la ejecución del código generado es 42 veces más rápido que ThunderSVM. Otro ejemplo es cuando el SVM se representa como una matriz densa, como los modelos creados para los conjuntos de datos SensIT, Connect-4, Poker y Satimage. La ejecución de esos modelos usando el código generado fue más rápida que ThunderSVM, en un rango de $24\,\%$ (SensIT) a $369\,\%$ (Satimage) más rápido.

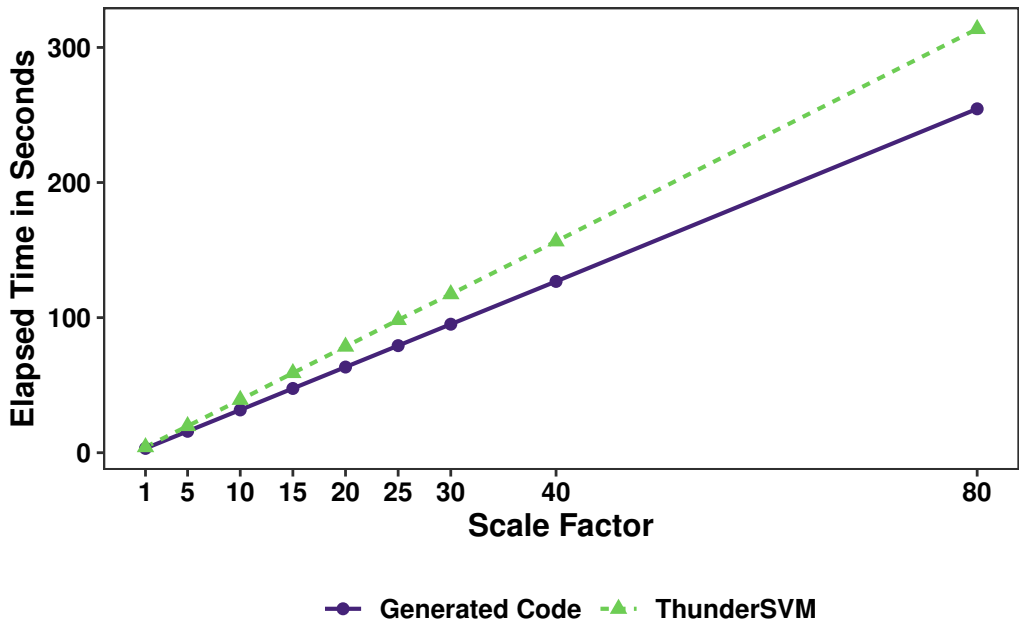


Figura 5.8: Gráfico de líneas de los resultados de la evaluación experimental del código generado y ThunderSVM utilizando la GPU para ejecutar el modelo SVM con cada factor de escala del conjunto de datos SensIT.

Escala	Total de observaciones	Código generado		ThunderSVM	
		Tiempo transcurrido en segundos	Observaciones procesadas por segundo	Tiempo transcurrido en segundos	Observaciones procesadas por segundo
1	67,557	2.27	29,726	3.56	18,974
10	675,570	22.54	29,969	33.38	20,237
20	1,351,140	44.94	30,068	66.87	20,207
40	2,702,280	89.85	30,074	132.74	20,357
80	5,404,560	179.85	30,050	265.87	20,328
100	6,755,700	224.76	30,057	333.63	20,249
200	13,511,400	449.74	30,042	667.90	20,230

Tabla 5.12: Resultados de los experimentos con modelos SVM multi-clase utilizando la GPU mediante el código generado y ThunderSVM para el conjunto de datos Connect-4.

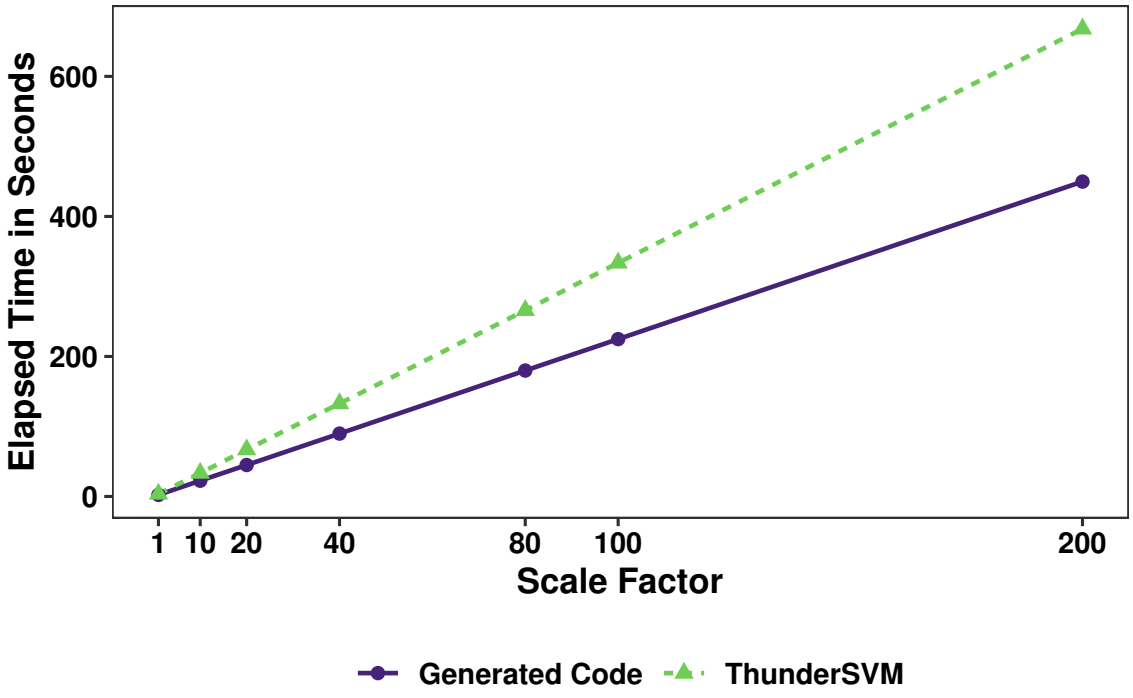


Figura 5.9: Gráfico de líneas de los resultados de la evaluación experimental del código generado y ThunderSVM utilizando la GPU para ejecutar el modelo SVM con cada factor de escala del conjunto de datos Connect-4.

		Código generado		ThunderSVM	
Escala	**Total de observaciones**	**Tiempo transcurrido en segundos**	**Observaciones procesadas por segundo**	**Tiempo transcurrido en segundos**	**Observaciones procesadas por segundo**
1	25,010	0.52	48,547	1.48	16,955
10	250,100	5.10	49,061	13.22	18,916
20	500,200	10.19	49,102	26.22	19,079
40	1,000,400	20.37	49,119	52.02	19,231
80	2,000,800	40.72	49,130	103.94	19,249
100	2,501,000	50.90	49,137	129.84	19,263
500	12,505,000	254.57	49,122	691.79	18,076
1000	25,010,000	508.97	49,138	N/A	N/A

Tabla 5.13: Resultados de los experimentos con modelos SVM multi-clase utilizando la GPU mediante el código generado y ThunderSVM para el conjunto de datos Poker.

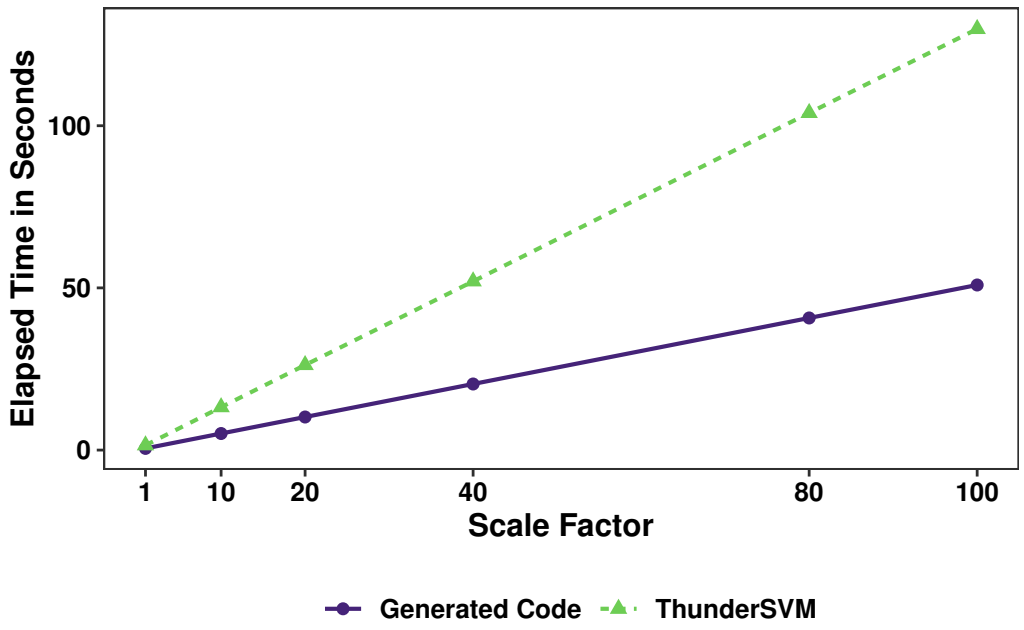


Figura 5.10: Gráfico de líneas del resultados de la evaluación experimental del código generado y ThunderSVM utilizando la GPU para ejecutar el modelo SVM con el factor de escala de hasta 100 del conjunto de datos de Poker.

		Código generado		ThunderSVM	
Escala	**Total de observaciones**	**Tiempo transcurrido en segundos**	**Observaciones procesadas por segundo**	**Tiempo transcurrido en segundos**	**Observaciones procesadas por segundo**
1	4,435	0.01	340,451	0.20	22,688
10	44,350	0.12	368,906	0.44	101,605
20	88,700	0.24	371,541	0.70	127,190
40	177,400	0.48	372,753	1.25	141,410
80	354,800	1.26	282,477	2.28	155,600
100	443,500	1.30	341,609	2.81	157,692
500	2,217,500	4.24	523,524	13.37	165,918
1000	4,435,000	7.02	631,655	26.71	166,013
1500	6,652,500	10.31	645,054	39.84	166,982
2000	8,870,000	13.90	638,041	54.20	163,650

Tabla 5.14: Resultados de los experimentos con modelos SVM multi-clase utilizando la GPU mediante el código generado y ThunderSVM para el conjunto de datos Satimage.

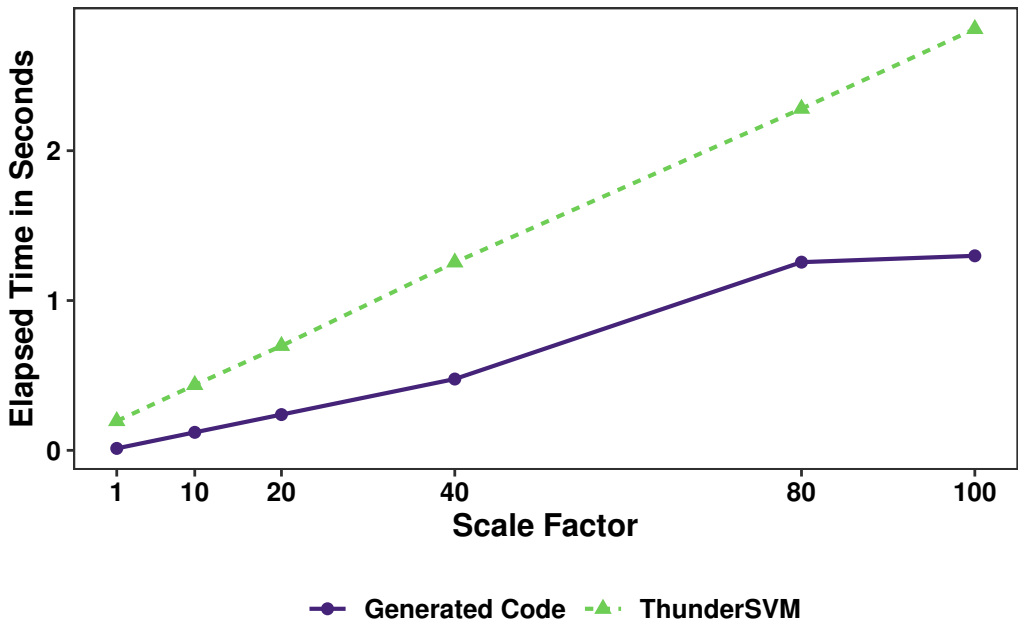


Figura 5.11: Gráfico de líneas de los resultados de la evaluación experimental del código generado y ThunderSVM usando la GPU para ejecutar el modelo SVM con los factores de escala 1, 10, 20, 40, 80 y 100 del conjunto de datos de Satimage.

CAPÍTULO 6
CONCLUSION

Esta disertación presenta el diseño y desarrollo de un compilador multi-destino para el despliegue de modelos predictivos inferidos con técnicas de *machine learning*. Esta investigación está motivada por la necesidad de abordar los siguientes dos problemas: 1) la automatización del despliegue de modelos predictivos aprendidos por máquina, y 2) la ejecución eficiente de modelos predictivos en entornos de producción. Hoy en día, existe un creciente interés en implementar modelos predictivos en aplicaciones del mundo real. Estas aplicaciones requieren ser rápidas, eficientes y robustas. Por lo tanto, la propuesta de esta disertación es de gran relevancia para cerrar la brecha entre la construcción de modelos predictivos y el desarrollo y mantenimiento de software de nivel de producción que integre esos modelos predictivos. Por lo general, el despliegue de modelos predictivos se realiza utilizando la misma herramienta de modelado donde se construyó el modelo. Este enfoque es apropiado cuando una herramienta analítica cumple los requisitos de un entorno de producción. Sin embargo, las herramientas de modelado no son un enfoque adecuado en un entorno de despliegue a escala industrial donde los modelos requieren un funcionamiento eficiente y auto-contenido.

Dentro de esta disertación hemos estudiado los principales enfoques para el despliegue de modelos predictivos. La principal contribución de esta disertación es el diseño e implementación del compilador multi-destino como una solución efectiva y eficiente para el despliegue de modelos predictivos. El compilador multi-destino propuesto genera código fuente a partir de descripciones formales de modelos predictivos. El código fuente generado está listo para integrarse en entornos de producción y permite automatizar la tarea de implementación en proyectos de aprendizaje automático. La propuesta de esta disertación es particularmente importante para ayudar a construir un módulo de *machine learning* de

un sistema. Por lo tanto, el compilador ofrece una solución con mucho potencial para disminuir la deuda técnica oculta. La codificación manual de modelos predictivos es una tarea costosa de realizar porque el conocimiento matemático que está detrás de los algoritmos de aprendizaje automático es complejo y generalmente está fuera del alcance de la experiencia de los ingenieros de software. Por la misma razón, su construcción y mantenimiento son difíciles. Los productos de software con capacidades predictivas deben estar libres de errores, ser eficientes y portátiles con dependencias mínimas.

La evaluación empírica extensa valida la viabilidad de esta propuesta. En nuestra implementación, hemos desarrollado un *front-end* para el estándar PMML. De esta manera, muchas herramientas analíticas compatibles con el estándar PMML se pueden usar para construir modelos predictivos para que los modelos se puedan implementar utilizando el compilador multi-destino propuesto. Los *back-end* implementados cubren diferentes lenguajes de destino como C/C ++ y Java. Ser capaz de implementar un modelo predictivo utilizando el código fuente, por ejemplo en C/C ++, permite una gestión de memoria más precisa. Además, el código compilado puede ejecutarse varias veces más rápido que el código interpretado. Mediante el uso de plantillas de los algoritmos de predicción, podemos identificar las instrucciones que se optimizarán para un lenguaje/arquitectura objetivo determinado. Por lo tanto, también podemos generar código fuente que se ejecute eficientemente en arquitecturas paralelas, como CPU o GPU multinúcleo. Nuestra propuesta reduce en gran medida el tiempo de implementación y con la evaluación experimental hemos demostrado la eficiencia del código fuente generado.

El compilador multi-destino traduce los modelos predictivos en código fuente para su implementación de manera automatizada. También mostramos que generamos código fuente listo para usar y que se ejecuta de manera eficiente sin edición manual. Esta es también una forma efectiva de integrar modelos predictivos en entornos de producción. El diseño robusto de nuestro compilador nos permite agregar fácilmente nuevos lenguajes destino según sea necesario.

Para futuros trabajos, planeamos implementar modelos de *deep learning* en el compilador multi-destino. Para lograr esto, requerimos desarrollar plantillas para la ejecución eficiente de modelos de *deep learning* de acuerdo con el lenguaje destino. También planeamos explorar la viabilidad del desarrollo de un *back-end* para generar código para FPGA. El uso de FPGA podría acelerar la ejecución de modelos predictivos que requieren cómputo intensivo.

REFERENCIAS

[1] J. G. Garrett Flynn, "Big Data: The BIG Factor Driving Competitive Advantage", KPMG, inf. téc., jun. de 2015.

[2] T. W. Miller, *Modeling Techniques in Predictive Analytics: Business Problems and Solutions with R*. Pearson Education, 2014.

[3] S. Lavalle, M. S. Hopkins, E. Lesser, R. Shockley y N. Kruschwits, "Analytics : The New Path to Value", IBM y MIT, inf. téc., 2010.

[4] E. Brynjolfsson, L. M. Hitt y H. H. Kim, "Strength in Numbers: How does Data-driven Decisionmaking Affect Firm Performance?", *SSRN eLibrary*, 2011.

[5] J. Wexler, W. Thompson y K. Aponte, "Time Is Precious, so are Your Models: SAS Provides Solutions to Streamline Deployment", SAS Institute Inc., inf. téc., 2013.

[6] J. Han, J. Pei y M. Kamber, *Data Mining: Concepts and Techniques*. Elsevier, 2011.

[7] V. Dhar, "Data Science and Prediction", *Communications of the ACM*, vol. 56, nº 12, 64–73, dic. de 2013.

[8] M. Kim, T. Zimmermann, R. DeLine y A. Begel, "The Emerging Role of Data Scientists on Software Development Teams", en *Proceedings of the International Conference on Software Engineering*, ACM, 2016, págs. 96-107.

[9] I. Goodfellow, Y. Bengio y A. Courville, *Deep Learning*. MIT press, 2016.

[10] S. J. Russell y P. Norvig, *Artificial Intelligence: A Modern Approach*, 2º ed. Pearson Education, 2003.

[11] U. Fayyad, D. Haussler y P. Stolorz, "KDD for Science Data Analysis: Issues and Examples", en *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, AAAI Press, 1996, págs. 50-56.

[12] SAS, *Sample, Explore, Modify, Model, and Assess (SEMMA)*, `https://documentation.sas.com`, [Online; accessed 10-October-2019], 2011.

[13] P. Chapman, J. Clinton, R. Kerber, T. Khabaza, T. Reinartz, C. Shearer y R. Wirth, "CRISP-DM 1.0 Step-by-step data mining guide", The CRISP-DM consortium, inf. téc., ago. de 2000.

[14] W. W. Eckerson, “Predictive Analytics”, Transforming Data With Intelligence, inf. téc., 2007, págs. 1-36.

[15] G. Piatetsky-Shapiro, *Data Mining Methodology Survey*, `https://www.kdnuggets.com/2014/10/crisp-dm-top-methodology-analytics-data-mining-data-science-projects.html`, [Online; accessed 19-July-2019], 2014.

[16] R. L. Grossman, “A Framework for Evaluating the Analytic Maturity of an Organization”, *International Journal of Information Management*, vol. 38, nº 1, págs. 45 -51, 2018.

[17] A. Guazzelli, M. Zeller, W.-C. Lin y G. Williams, “PMML: An Open Standard for Sharing Models”, *The R Journal*, vol. 1, nº 1, págs. 60-65, 2009.

[18] J. Pivarski, *PFA: Portable Format for Analytics (version 0.6)*, 2015.

[19] Y. Lee, A. Scolari, B.-G. Chun, M. Weimer y M. Interlandi, “From the Edge to the Cloud: Model Serving in ML.NET”, *Data Engineering*, vol. 41, nº 4, págs. 46-53, 2018.

[20] N. Rasiwasia, “Perspectives on Becoming an Applied Machine Learning Scientist”, *Computer*, vol. 52, nº 5, págs. 40-47, 2019.

[21] D. Merkel, “Docker: Lightweight Linux Containers for Consistent Development and Deployment”, *Linux Journal*, vol. 2014, nº 239, 2014.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot y E. Duchesnay, “Scikit-learn: Machine Learning in Python”, *Journal of Machine Learning Research*, vol. 12, págs. 2825-2830, nov. de 2011.

[23] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian y S. Vaithyanathan, “SystemML: Declarative Machine Learning on MapReduce”, en *Proceedings of the International Conference on Data Engineering*, 2011, págs. 231-242.

[24] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve y S. Tatikonda, “SystemML: Declarative Machine Learning on Spark”, *Proceedings of the VLDB Endowment*, vol. 9, nº 13, págs. 1425-1436, sep. de 2016.

[25] S. Chan, T. Stone, K. P. Szeto y K. H. Chan, “PredictionIO: A Distributed Machine Learning Server for Practical Software Development”, en *Proceedings of the ACM*

International Conference on Information and Knowledge Management, New York, NY, USA: ACM, 2013, págs. 2493-2496.

[26] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen y col., "MLlib: Machine Learning in Apache Spark", *The Journal of Machine Learning Research*, vol. 17, nº 1, págs. 1235-1241, 2016.

[27] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin y B. Recht, "KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics", en *Proceedings of the IEEE International Conference on Data Engineering*, 2017, págs. 535-546.

[28] E. Ma, V. Gupta, M. Hsu y I. Roy, "Dmapply: A Functional Primitive to Express Distributed Machine Learning Algorithms in R", *Proceedings of the VLDB Endowment*, vol. 9, nº 13, págs. 1293-1304, sep. de 2016.

[29] A. Kunft, L. Stadler, D. Bonetta, C. Basca, J. Meiners, S. Breß, T. Rabl, J. Fumero y V. Markl, "ScootR: Scaling R Dataframes on Dataflow Systems", en *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA: ACM, 2018, págs. 288-300.

[30] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi y K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine", *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, nº 4, 2015.

[31] R. M. Deak y J. H. Morra, "Aloha: A Machine Learning Framework for Engineers", en *Proceedings of the SysML Conference*, 2018.

[32] N. Talagala, S. Sundararaman, V. Sridhar, D. Arteaga, Q. Luo, S. Subramanian, S. Ghanta, L. Khermosh y D. Roselli, "ECO: Harmonizing Edge and Cloud with ML/DL Orchestration", en *Proceedings of the USENIX Workshop on Hot Topics in Edge Computing*, Boston, MA: USENIX Association, 2018.

[33] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker y I. Stoica, "Apache Spark: A Unified Engine for Big Data Processing", *Communications of the ACM*, vol. 59, nº 11, págs. 56-65, oct. de 2016.

[34] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu y X. Zheng, "TensorFlow: A System for Large-scale Machine Learning", en *Proceedings of the Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA: USENIX Association, 2016, págs. 265-283.

[35] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer y M. Interlandi, "PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems", en *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2018, págs. 611-626.

[36] L. B. Miguel, D. Takabayashi, J. R. Pizani, T. Andrade y B. West, "Marvin - Open Source Artificial Intelligence Platform", en *Proceedings of the International Conference on Predictive Applications and APIs*, C. Hardgrove, L. Dorard y K. Thompson, eds., ép. Proceedings of Machine Learning Research, vol. 82, Microsoft NERD, Boston, USA: PMLR, 2018, págs. 33-44.

[37] W. Wang, J. Gao, M. Zhang, S. Wang, G. Chen, T. K. Ng, B. C. Ooi, J. Shao y M. Reyad, "Rafiki: Machine Learning as an Analytics Service System", en *Proceedings of the VLDB Endowment*, vol. 12, oct. de 2018, págs. 128-140.

[38] A. Ronacher, *Flask (A Python Microframework)*, `https://flask.palletsprojects.com`, [Online; accessed 19-July-2019], 2011.

[39] Data Mining Group, *Predictive Model Markup Language*, `http://dmg.org/pmml/v4-4/GeneralStructure.html`, [Online; accessed 19-July-2019], 2019.

[40] Data Mining Group, *Portable Format for Analytics*, `http://dmg.org/pfa/index.html`, [Online; accessed 19-July-2019], 2019.

[41] ONNX, *Open Neural Networks Exchange Format*, `http://onnx.ai/`, [Online; accessed 19-July-2019], 2019.

[42] R. Grossman, S. Bailey, A. Ramu, B. Malhi, P. Hallstrom, I. Pulleyn y X. Qin, "The Management and Mining of Multiple Predictive Models Using the Predictive Modeling Markup Language", *Information and Software Technology*, vol. 41, nº 9, págs. 589 -595, 1999.

[43] J. Chaves, C. Curry, R. L. Grossman, D. Locke y S. Vejcik, "Augustus: the Design and Architecture of a PMML-based Scoring Engine", en *Proceedings of the international workshop on Data mining standards, services and platforms*, ACM, 2006, págs. 38-46.

[44] D. Gorea, "Knowledge as a Service. An Online Scoring Engine Architecture", en *Proceedings of the International Multi-Conference on Computing in the Global Information Technology*, 2008, págs. 1-6.

[45] P. Nathan y G. Kathalagiri, "Pattern: PMML for Cascading and Hadoop", en *Proceedings of the Workshop on Predictive Markup Langugage Modeling*, 2013.

[46] Openscoring, *JPMML*, `http://openscoring.io/`, [Online; accessed 19-July-2019], 2019.

[47] P. Tendick y A. Mockus, "Decisions as a Service for Application Centric Real Time Analytics", en *Proceedings of the International Workshop on Big Data Software Engineering*, 2016, págs. 1-7.

[48] J. Heit, J. Liu y M. Shah, "An Architecture for the Deployment of Statistical Models for the Big Data Era", en *Proceedings of the IEEE International Conference on Big Data*, 2016, págs. 1377-1384.

[49] J. Pivarski, C. Bennett y R. L. Grossman, "Deploying Analytics with the Portable Format for Analytics (PFA)", en *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, 2016, págs. 579-588.

[50] D. Vohra, "Apache Avro", en *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*. Berkeley, CA: Apress, 2016, págs. 303-323.

[51] Microsoft, *ONNX Runtime*, `https://github.com/microsoft/onnxruntime`, [Online; accessed 19-July-2019], 2019.

[52] A. Team, "AzureML: Anatomy of a Machine Learning Service", en *Proceedings of The International Conference on Predictive APIs and Apps*, L. Dorard, M. D. Reid y F. J. Martin, eds., ép. Proceedings of Machine Learning Research, vol. 50, Sydney, Australia: PMLR, 2016, págs. 1-13.

[53] A. Bhattacharjee, Y. Barve, S. Khare, S. Bao, A. Gokhale y T. Damiano, "Stratum: A Serverless Framework for the Lifecycle Management of Machine Learning-based Data Analytics Tasks", en *Proceedings of the USENIX Conference on Operational Machine Learning*, 2019, págs. 59-61.

[54] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez y I. Stoica, "Clipper: A Low-latency Online Prediction Serving System", en *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2017, págs. 613-627.

[55] J. Zhao, T. Tiplea, R. Mortier, J. Crowcroft y L. Wang, "Data Analytics Service Composition and Deployment on Edge Devices", en *Proceedings of the Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*, New York, NY, USA: ACM, 2018, págs. 27-32.

[56] ——, "Data Analytics Service Composition and Deployment on IoT Devices", en *Proceedings Annual International Conference on Mobile Systems, Applications, and Services*, New York, NY, USA: ACM, 2018, págs. 502-504.

[57] A. Madhavapeddy y D. J. Scott, “Unikernels: The Rise of the Virtual Library Operating System”, *Communications of the ACM*, vol. 57, nº 1, págs. 61-69, ene. de 2014.

[58] S. Zhao, M. Talasila, G. Jacobson, C. Borcea, S. A. Aftab y J. F. Murray, “Packaging and Sharing Machine Learning Models via the Acumos AI Open Platform”, en *Proceedings of the International Conference on Machine Learning and Applications*, 2018, págs. 841-846.

[59] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe y col., “Accelerating the Machine Learning Lifecycle with MLflow”, *Data Engineering*, vol. 41, nº 4, págs. 39-45, 2018.

[60] R. Chard, L. Ward, Z. Li, Y. Babuji, A. Woodard, S. Tuecke, K. Chard, B. Blaiszik y I. Foster, “Publishing and Serving Machine Learning Models with DLHub”, en *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, New York, NY, USA: ACM, 2019, págs. 1-7.

[61] S. Balkan y M. Goul, “Advances in Predictive Modeling: how In-database Analytics will Evolve to Change the Game”, *Business Intelligence Journal*, vol. 15, nº 2, págs. 17-25, 2010.

[62] J. Clear, D. Dunn, B. Harvey, M. Heytens, P. Lohman, A. Mehta, M. Melton, L. Rohrberg, A. Savasere, R. Wehrmeister y M. Xu, “NonStop SQL/MX Primitives for Knowledge Discovery”, en *Proceedings of the SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, USA: ACM, 1999, págs. 425-429.

[63] A. Hinneburg, D. Habich y W. Lehner, “Combi-Operator Database Support for Data Mining Applications”, en *Proceedings of the International Conference on Very Large Data Bases*, J.-C. Freytag, P. Lockemann, S. Abiteboul, M. Carey, P. Selinger y A. Heuer, eds., San Francisco: Morgan Kaufmann, 2003, págs. 429 -439.

[64] A. Netz, S. Chaudhuri, U. Fayyad y J. Bernhardt, “Integrating Data Mining with SQL Databases: OLE DB for Data Mining”, en *Proceedings of the International Conference on Data Engineering*, 2001, págs. 379-387.

[65] C. Ordonez, “Statistical Model Computation with UDFs”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, nº 12, págs. 1752-1765, 2010.

[66] M. L. Koc y C. Ré, “Incrementally Maintaining Classification Using an RDBMS”, *Proceedings of the VLDB Endowment*, vol. 4, nº 5, págs. 302-313, feb. de 2011.

[67] C. Ordonez y Z. Chen, "Horizontal Aggregations in SQL to Prepare Data Sets for Data Mining Analysis", *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, nº 4, págs. 678-691, 2012.

[68] X. Feng, A. Kumar, B. Recht y C. Ré, "Towards a Unified Architecture for in-RDBMS Analytics", en *Proceedings of the ACM SIGMOD International Conference on Management of Data*, New York, NY, USA: ACM, 2012, págs. 325-336.

[69] K. K. Das, E. Fratkin, A. Gorajek, K. Stathatos y M. Gajjar, "Massively Parallel In-database Predictions Using PMML", en *Proceedings of the Workshop on Predictive Markup Language Modeling*, ACM, 2011, págs. 22-27.

[70] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li y A. Kumar, "The MADlib Analytics Library: Or MAD Skills, the SQL", *Proceedings of the VLDB Endowment*, vol. 5, nº 12, págs. 1700-1711, ago. de 2012.

[71] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker y I. Stoica, "Shark: SQL and Rich Analytics at Scale", en *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ép. SIGMOD '13, New York, NY, USA: ACM, 2013, págs. 13-24.

[72] S. Prasad, A. Fard, V. Gupta, J. Martinez, J. LeFevre, V. Xu, M. Hsu y I. Roy, "Large-scale Predictive Analytics in Vertica: Fast Data Transfer, Distributed Model Creation, and In-database Prediction", en *Proceedings of the SIGMOD International Conference on Management of Data*, ép. SIGMOD '15, New York, NY, USA: ACM, 2015, págs. 1657-1668.

[73] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar y H. Esmaeilzadeh, "In-RDBMS Hardware Acceleration of Advanced Analytics", *Proceedings of the VLDB Endowment*, vol. 11, nº 11, págs. 1317-1331, jul. de 2018.

[74] M. E. Schüle, M. Bungeroth, A. Kemper, S. Günnemann y T. Neumann, "MLearn: A Declarative Machine Learning Language for Database Systems", en *Proceedings of the International Workshop on Data Management for End-to-End Machine Learning*, 2019.

[75] P. Andrews, A. Kalro, H. Mehanna y A. Sidorov, "Productionizing Machine Learning Pipelines at Scale", en *Procedings of the ML Systems Workshop at International Conference on Machine Learning*, 2016.

[76] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong y X. Wang, "Applied Machine Learning at Facebook: A Datacenter Infras-

tructure Perspective", en *Proceedings of the International Symposium on High Performance Computer Architecture*, 2018, págs. 620-629.

[77] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. M. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo y P. Zhang, "Machine Learning at Facebook: Understanding Inference at the Edge", en *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2019, págs. 331-344.

[78] L. E. Li, E. Chen, J. Hermann, P. Zhang y L. Wang, "Scaling Machine Learning as a Service", en *Proceedings of the International Conference on Predictive Applications and APIs*, 2017, págs. 14-29.

[79] J. Hermann y M. Del Balso, *Meet Michelangelo: Uber's Machine Learning Platform*, Online, Accessed August-2019, 2017.

[80] M. Barbareschi, A. Mazzeo y A. Vespoli, "Malicious Traffic Analysis on Mobile Devices: a Hardware Solution", *International Journal of Big Data Intelligence*, vol. 2, nº 2, págs. 117-126, 2015.

[81] S. Gopinath, N. Ghanathe, V. Seshadri y R. Sharma, "Compiling KB-sized Machine Learning Models to Tiny IoT Devices", en *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA: ACM, 2019, págs. 79-95.

[82] A. V. Aho, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-wesley Reading, 2007, vol. 2.

[83] W. M. Waite y G. Goos, *Compiler Construction*. Springer Science & Business Media, 2012.

[84] E. Ilyushin y D. Namiot, "On Source-to-Source Compilers", *International Journal of Open Information Technologies*, vol. 4, nº 5, págs. 48-51, 2016.

[85] A. W. Appel, *Modern Compiler Implementation in C*. Cambridge university press, 2004.

[86] K. Cooper y L. Torczon, *Engineering a Compiler*. Elsevier, 2011.

[87] U. Olsson, *Generalized Linear Models: An Applied Approach*. Studentlitteratur Lund, 2002.

[88] W. H. Press, S. A. Teukolsky, W. T. Vetterling y B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge university press, 2007.

[89] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[90] C. Cortes y V. Vapnik, "Support-Vector Networks", *Machine Learning*, vol. 20, nº 3, págs. 273-297, sep. de 1995.

[91] J. Palsberg y C. B. Jay, "The Essence of the Visitor Pattern", en *Proceedingsof the International Computer Software and Applications Conference*, 1998, págs. 9-15.

[92] R. Ramakrishnan y J. Gehrke, *Database Management Systems*. McGraw Hill, 2000.

[93] L. Dagum y R. Menon, "OpenMP: an Industry Standard API for Shared-Memory Programming", *IEEE Computational Science and Engineering*, vol. 5, nº 1, págs. 46-55, 1998.

[94] J. Nickolls, I. Buck, M. Garland y K. Skadron, "Scalable Parallel Programming with CUDA", en *Proceedings of the International Conference and Exhibition on Computer Graphics and Interactive Techniques*, New York, NY, USA: ACM, 2008, 16:1-16:14.

[95] V. Volkov y J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra", en *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, págs. 1-11.

[96] I. Yeh y C. Lien, "The Comparisons of Data Mining Techniques for the Predictive Accuracy of Probability of Default of Credit Card Clients", *Expert Systems with Applications*, vol. 36, nº 2, Part 1, págs. 2473 -2480, 2009.

[97] V. Arzamasov, K. Böhm y P. Jochem, "Towards Concise Models of Grid Stability", en *Proceedings of the International Conference on Communications, Control, and Computing Technologies for Smart Grids*, 2018, págs. 1-6.

[98] C. O. Sakar, S. O. Polat, M. Katircioglu y Y. Kastro, "Real-time Prediction of Online Shoppers' Purchasing Intention Using Multilayer Perceptron and LSTM Recurrent Neural Networks", *Neural Computing and Applications*, vol. 31, nº 10, págs. 6893-6908, 2018.

[99] Y. Lecun, L. Bottou, Y. Bengio y P. Haffner, "Gradient-based Learning Applied to Document Recognition", *Proceedings of the IEEE*, vol. 86, nº 11, págs. 2278-2324, 1998.

[100] D. D. Lewis, Y. Yang, T. G. Rose y F. Li, "Rcv1: A New Benchmark Collection for Text Categorization Research", *Journal of machine learning research*, vol. 5, nº Apr, págs. 361-397, 2004.

[101] R. Bekkerman y M. Scholz, "Data Weaving: Scaling up the State-of-the-art in Data Clustering", en *Proceedings of the ACM conference on Information and knowledge management*, ACM, 2008, págs. 1083-1092.

[102] M. F. Duarte y Y. H. Hu, "Vehicle Classification in Distributed Sensor Networks", *Journal of Parallel and Distributed Computing*, vol. 64, nº 7, págs. 826-838, 2004.

[103] D. Dua y C. Graff, *UCI machine learning repository*, 2017.

[104] R. Cattral, F. Oppacher y D. Deugo, "Evolutionary Data Mining with Automatic Rule Generalization", en *Recent Advances in Computers, Computing and Communications*, 2002, págs. 296-300.

[105] M. Kuhn, with contributions from: Jed Wing, S. Weston, A. Williams, C. Keefer, A. Engelhardt, T. Cooper, Z. Mayer, B. Kenkel, the R Core Team, M. Benesty, R. Lescarbeau, A. Ziem, L. Scrucca, Y. Tang, C. Candan y T. Hunt, *caret: Classification and Regression Training*, R package version 6.0-76, 2017.

[106] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2018.

[107] D. Meyer, E. Dimitriadou, K. Hornik, A. Weingessel y F. Leisch, *e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien*, R package version 1.6-8, 2017.

[108] G. Williams, T. Jena, W. C. Lin, M. Hahsler, Z. Inc, H. Ishwaran, U. B. Kogalur, R. Guha y D. Bolotov, *pmml: Generate PMML for Various Models*, R package version 1.5.2, 2017.

[109] Z. Wen, J. Shi, Q. Li, B. He y J. Chen, "ThunderSVM: A fast SVM library on GPUs and CPUs", *The Journal of Machine Learning Research*, vol. 19, nº 1, págs. 797-801, 2018.