

Universidad Nacional de Luján

Licenciatura en Sistemas de Información



Políticas de admisión a caché gestionadas
mediante árboles de decisión adaptativos

*Tesina presentada para aplicar al título
de Licenciado en Sistemas de Información*

Tonin Monzón Francisco

Director: Banchero, Santiago
Codirector: Tolosa, Gabriel Hernán

Mayo 2018

1. Resumen

Millones de consultas son realizadas diariamente a los motores de búsqueda web, donde la utilización de memoria caché es crucial para reducir el tiempo de respuesta y aumentar el rendimiento. En la literatura, diversos autores han propuesto la utilización de técnicas de aprendizaje de máquina para aumentar la eficiencia de la caché. Hasta el momento, los trabajos en el área consistieron en la utilización de algoritmos con funcionamiento por lote para gestionar las políticas de admisión y/o reemplazo en diferentes niveles de caché. Estos algoritmos construyen modelos estáticos que reducen su efectividad ante cambios en el comportamiento de los usuarios. Por otro lado, la investigación en el área de minería de flujos de datos ha aportado nuevos algoritmos, sistemas y plataformas para hacer frente a ambientes con generación continua de datos, altas tasas de arribo y elevados niveles de fluctuación en ellas, características que a su vez definen a la resolución de consultas en motores de búsqueda web. Basado en lo anterior, en este trabajo se propone la utilización del algoritmo Hoeffding Adaptive Tree, específicamente desarrollado para el aprendizaje adaptativo, para generar reglas que permitan predecir futuras apariciones de las consultas. El rendimiento del mismo es comparado contra C4.5, un algoritmo clásico con funcionamiento por lote. En los experimentos realizados, Hoeffding Adaptive Tree permite duplicar los aciertos de la caché que obtiene C4.5. Complementariamente, se diseña una solución sobre el motor de procesamiento de streaming Apache Storm, que proporciona al árbol de decisión adaptativo con los ejemplos necesarios para mantenerse actualizado.

*Ningún mar en calma
hizo experto a un marinero...*

Me gustaría dedicar unas líneas a algunas de las personas que hicieron posible llevar a cabo este trabajo.

Al Mg. Santiago Banchero, director de este trabajo, quiero agradecerle por su enorme predisposición para seguir mis avances. Su apoyo y motivación fueron decisivos para el desarrollo de todo el trabajo.

Al Dr. Gabriel Tolosa, por sus fundamentales aportes. Su experiencia y conocimiento permitieron sortear obstáculos que se presentaron en diversas etapas de este trabajo.

Sin dudas, me siento muy agradecido de haber contado con profesionales de este calibre en la carrera y más aún, de que sean ellos mis tutores.

Gracias a mi compañero de trabajo y amigo, el Lic. Alejandro Iglesias, por sus consejos y recomendaciones.

A mi madre Carolina, a quien siempre le estaré agradecido por sus charlas motivadoras desde la niñez. Debido a su consejo escogí formarme en esta casa; recomendación que le agradeceré eternamente.

A mi padre Mario, gracias a un gran esfuerzo de su parte conté con los recursos necesarios para llevar a delante mi formación. Mediante el ejemplo me inculcó el trabajo constante y la tenacidad que me llevaron a alcanzar este punto.

A mi hermano mellizo Adolfo, por todos sus consejos y aliento. Sin dudas, ha sido siempre un referente para mi.

A Eliana, mi novia, amiga y compañera con la que tantas veces nos juntamos a estudiar, cada uno sobre nuestras respectivas carreras, para no perder oportunidad de estar cerca. Le agradezco su infinito apoyo y por confiar siempre en mi.

Y finalmente, me gustaría agradecer a mi abuela, Purísima, quien me acompañó con incontables mates durante la carrera. Sé que desde algún lugar festeja verme alcanzar mis sueños.

*A mi querida Abuela,
Purísima Martínez,
Por su amor incondicional...*

1. Resumen	2
2. Introducción	6
2.1. Objetivos de este trabajo	10
2.2. Contribuciones	11
2.3. Organización	11
3. Antecedentes y trabajos relacionados	13
3.1. Políticas de admisión a cache en motores de búsqueda.	13
3.1.1. Introducción	13
3.1.2. Arquitectura de los motores de búsqueda	13
3.2. Políticas de caché	15
3.2.1. Niveles de jerarquía de caché	16
3.2.2. Trabajos relacionados	17
3.3. Árboles de decisión	19
3.3.1. Algoritmos de árboles de decisión tradicionales	19
3.3.2. Hoeffding Tree	22
3.3.2.1. Manejo de atributos numéricos	24
3.3.2.1.1. VMFL	24
3.3.2.1.2. Árbol binario exhaustivo	25
3.3.2.1.3. Resumen de Quantiles	25
3.3.2.1.4. Aproximación Gaussiana	26
3.3.3. Concept-Adapting very Fast Decision Tree	26
3.3.4. Hoeffding Adaptive Tree	28
3.4. Stream Processing Engines	29
3.4.1. Introducción	29
3.4.2. Apache Storm	29
3.4.2.1. Elementos principales de Apache Storm	30
4. Metodología y propuesta	31
4.1. Preprocesamiento y Análisis exploratorio del conjunto de datos	31
4.1.1. Introducción	31

4.1.2. Preprocesamiento	32
4.1.2.1. Datos faltantes	34
4.1.2.2. Detección de palabras con errores de escritura	34
4.1.3. Exploración de los datos	35
4.1.3.1. Sesiones de búsqueda	41
4.1.3.1.1. Cantidad de consultas realizadas	41
4.1.3.1.2. Cantidad de resultados seleccionados en la sesión	43
4.1.3.1.3. Cantidad de resultados seleccionados en la primera posición del ranking	45
4.1.3.1.4. Atributos de frecuencia	46
4.1.3.1.5. Tasas de consultas	46
4.2. Descripción de la solución propuesta	48
5. Experimentos y Resultados	52
5.1. Definición de métricas	52
5.2. Valor de referencia	54
5.3. Desarrollo del modelo con Hoeffding Adaptive Tree	55
5.3.1. Puesta a punto	55
5.3.1.1. Evaluación de criterios de división	56
5.3.1.2. Evaluación de métodos para manejo de atributos numéricos	56
5.3.4. Comparación de rendimiento de C4.5 y Hoeffding Adaptive Tree	57
5.3.4.1. Utilización de recursos	60
5.3.4.2. Rendimiento frente a cambios de concepto	62
5.3.4.3. Evaluación del modelo en combinación con políticas de desalojo dinámicas	64
6. Conclusiones y trabajos futuros	73
6.1. Trabajos futuros	74
7. Bibliografía	75

2. Introducción

Desde el surgimiento de los primeros motores de búsqueda en los años 90' el crecimiento en cantidad de usuarios no ha hecho más que incrementarse. Aproximadamente el 90% de ellos utilizan motores de búsqueda para obtener información [1]. Al hacer uso de ellos, los usuarios tienen grandes expectativas respecto a la calidad de sus resultados y al tiempo en el que le son entregados [2]. La web indexable (contenido accesible por los motores de búsqueda) es cada vez más grande, más de 45 mil millones de páginas según [3], por lo que resolver una consulta requiere el procesamiento de un inmenso volumen de datos. Otra dificultad que complejiza las operaciones son las altas tasas de consultas que arriban de manera ininterrumpida provocando que cualquier actividad complementaria o de mantenimiento deba hacerse, en lo posible, en tiempo real. Estas últimas características, asociadas a los motores de búsqueda, son referidas en la literatura como volumen y velocidad, dos de las dimensiones que caracterizan un problema de grande datos (*big data*) [4,5] .

La arquitectura estándar utilizada para dar soporte a la resolución de consultas se ejecuta sobre un grupo de computadoras interconectadas por una red de área local (*LAN*) [6], configuración que normalmente es denominada *cluster*. Se pueden diferenciar dos tipos de nodos que componen este sistema, un nodo intermediario (*broker*) cuya tarea consiste en proveer la interfaz de usuario y asignar una parte del trabajo al segundo componente, los nodos de búsqueda (*search nodes*), en base a algún criterio definido que suele corresponderse con la optimización. Los search nodes resuelven las consultas y devuelve el resultado al broker.

Uno de los mecanismos más utilizados para la optimización de la resolución de consultas es el uso de cache, que consiste básicamente en mantener en una memoria rápida elementos previamente utilizados. Decidir los criterios por los cuales un elemento es agregado a la memoria caché (políticas de admisión) y por los cuales es removida de ella (políticas de desalojo), es crucial para la eficiencia del método. Tradicionalmente éstas políticas se basaban en criterios de frecuencia, tiempo de aparición o costo [7].

En la actualidad diversos autores [7–9] han propuesto, como alternativa a los criterios tradicionales, la utilización de algoritmos de aprendizaje de máquina (*Machine Learning*, de ahora en más ML) para decidir qué elementos deben ser admitidos y cuales desalojadas de la memoria caché en base a diversos atributos de los mismos (*featured based cache*). Aun en los trabajos más recientes en los que se aplica ML, los modelos se entrena en modo por lotes. Este modo estático limita a los modelos de predicción a mantenerse ajustados a solo un periodo finito

de tiempo. En el entorno de motor de búsqueda la frecuencia de aparición de las consultas evoluciona, los usuarios expresan un aumento en su interés en consultas relacionadas con eventos recientes, llevando a incrementos en la frecuencia de estas consultas [10]. Por este motivo, un modelo ideal con el cual predecir si una consulta volverá a repetirse debe actualizarse en el tiempo, capturando las nuevas tendencias y olvidando aquellas antiguas que ya no están en vigencia.

El aprendizaje de máquina es un campo de las ciencias de la computación que intenta darle a una máquina la habilidad de mejorar su eficiencia, sin haber sido programado específicamente para tal tarea [11]. La entrada de estos algoritmos de aprendizaje es un conjunto ejemplos, más específicamente denominados instancias, que son independientes entre sí. Una instancia se caracteriza por los valores de un conjunto predeterminado de atributos que pueden tener valores numéricos o nominales. En este campo se distinguen tres tipos de problemas de aprendizaje: supervisado, no supervisado y aprendizaje por refuerzo [12]. Los algoritmos de aprendizaje de máquina supervisados intentan generar una hipótesis a partir de las instancias vistas con las cuales hacer predicciones de las futuras instancias [13]. El aprendizaje no supervisado implica encontrar patrones en datos de entrada sin que sea suministrada una salida esperada para ellos. El aprendizaje por refuerzo consiste en usar recompensas observadas para desarrollar una política óptima (o cercana a la óptima) para un entorno[12].

Los algoritmos de aprendizaje de máquina se suelen utilizar en conjunto con otras técnicas en un proceso que se denomina minería de datos. Formalmente, este concepto consiste en la aplicación de algoritmos específicos para extraer patrones a partir de grandes conjuntos de datos combinando áreas de investigación, como aprendizaje de máquina, reconocimiento de patrones, bases de datos, estadística, visualización de datos y computación de alta performance (*High Performance Computing*) [14]. Ésta es a su vez considerada una de las etapas del proceso de descubrimiento de conocimiento, cuyo producto final es conocimiento útil para ser incorporado a un sistema o simplemente ser reportado y/o documentado [15].

Previo a la aplicación de técnicas de minería de datos, el acercamiento clásico consiste en una estrategia manual, en el que un analista debe familiarizarse con los datos y a partir de ello detectar patrones. Realizado de esta forma, resulta un proceso lento, costoso y altamente subjetivo, falto de escalabilidad y de aplicación inviable para los volúmenes de datos que son procesados por las aplicaciones modernas. En las pasadas tres décadas, la práctica e investigación en aprendizaje automático se han focalizado en un funcionamiento en modo por

lotes, usando pequeños *datasets* con un conjunto de entrenamiento completamente disponible para el algoritmo que aprende de él [16]. Sin embargo, las necesidades de aplicaciones de descubrimiento de conocimiento continúan en un proceso de evolución, y como ha sido presentado previamente la necesidad de mantener estos sistemas actualizados es cada vez mayor. Los datos, en muchas de las aplicaciones actuales, provienen de un flujo continuo e infinito (*stream de datos*). Frente a los streams de datos los sistemas convencionales de descubrimiento de conocimiento encuentran grandes limitaciones para adaptarse con la velocidad y frecuencia que estos ambientes naturalmente dinámicos requieren.

Las redes sociales, las transacciones electrónicas, redes de sensores y los sistemas de geolocalización, junto con los motores de búsqueda son ejemplos de sistemas que producen grandes volúmenes de datos con gran rapidez, que desafían las infraestructuras y las técnicas de minería de datos actuales para extraer conocimiento. Esto da lugar a que se desarrollen técnicas, algoritmos y sistemas específicos para el tratamiento de streaming, un proceso que actualmente se lo conoce como minería de streaming (*data stream mining*).

Desde hace algunos años el interés en la investigación y desarrollo de software para tratamiento streaming se ha incrementado, y se han desarrollado algoritmos de aprendizaje de máquina para diversas problemáticas como clustering, clasificación y minería de secuencias [17].

En estos sistemas el conjunto de datos de entrenamiento cuenta con algunas de las siguientes propiedades que complejizan su tratamiento:

- El dataset completo no está disponible al momento de la construcción del modelo de predicción.
- El tamaño del dataset es demasiado grande para que se genere en memoria un modelo a partir de él.
- Las propiedades estadísticas del dataset evolucionan a través del tiempo.

Idealmente se espera que un algoritmo que aprende de streaming pueda operar de forma continua e indefinida, aprendiendo de nuevos ejemplos a medida que llegan y que nunca se pierda información potencialmente importante [18]. Es decir, que en un sistema de datos que implemente aprendizaje de máquina por flujos, la diferencia que tradicionalmente existía entre una etapa de entrenamiento y otra de ejecución se diluye, constituyéndose como un continuo de ejecución y aprendizaje. Los requerimientos para aquellos algoritmos que funcionan en ambientes de streams de datos pueden resumirse en [19]:

- Leer cada instancia de entrenamiento como máximo una vez.

- Usar una cantidad limitada de memoria.
- Funcionar en un tiempo limitado.
- Tener el modelo listo para predecir en cualquier momento, independiente del orden las instancias vistas.

La mayoría de las aplicaciones de descubrimiento de conocimiento convencionales tienen un cuello de botella en lo que respecta a tiempo y memoria utilizados. Habitualmente, cuando se trabaja en modo por lotes, el obstáculo de contar con memoria limitada en el aprendizaje y teniendo una cantidad de datos finitos es superado produciendo arreglos aleatorios de conjuntos de entrenamiento y prueba. En el contexto de streaming, desarrollar modelos a partir de un conjunto potencialmente infinito de datos implica nuevos desafíos. Además de los requerimientos mencionados, los algoritmos deben tener mecanismos para incorporar cambios de concepto, siendo capaces de olvidar datos desactualizados y ajustarse al estado más reciente de ellos [16].

Los árboles decisión son uno de los algoritmos más utilizados en minería de datos para clasificación, principalmente por su alto grado de interpretabilidad [16]. En el año 2000, Domingos y Hulten [20] proponen Hoeffding tree, un algoritmo de árboles de decisión que es capaz de entrenar un modelo de predicción a partir de un stream de datos, en un tiempo constante y sin la necesidad de que las instancias de entrenamiento estén almacenadas. A partir de este algoritmo, diversos autores desarrollaron mejoras. Una de ellas, ideada por Bifet y Gavaldá [21], incorpora la capacidad de adaptarse a cambios de concepto en las distribuciones de los datos, con reducido uso de recursos. Este algoritmo se conoce como Hoeffding Adaptive Tree (HAT) y es el estado del arte en algoritmos para minería de streaming. Por tal motivo, es utilizado en este trabajo para generar un modelo de predicción que de soporte a las políticas de admisión a caché.

En los últimos años, se han desarrollado una serie de plataformas para procesamiento de streaming de datos denominados Stream Processing Engines (SPE). Habitualmente los sistemas que requieren procesar grandes volúmenes de datos y en tiempo real utilizaban código específico. Los SPE permiten abordar este tipo de desarrollos en lenguajes de propósito general como C++ o Java, incluyendo además abstracciones de funcionalidades necesarias en un software de este tipo (Manejo de errores, registro de actividades, sincronización de tareas). Por lo tanto, el uso de un SPE simplifica el desarrollo software para el procesamiento de streaming derivando en la reducción de tiempo de desarrollo y costo de mantenimiento [22].

Por su parte, la variante HAT, es un clasificador adecuado para generalizar un modelo que se mantenga actualizado, capaz de predecir ininterrumpidamente si una consulta volverá a repetirse en un futuro cercano. A partir de la predicción del modelo se determina qué consultas deberían almacenarse en caché. Todas las tareas complementarias a la predicción, como por ejemplo, la transformación de datos o lectura de registros, deben hacerse acorde a los exigencias que son requeridas para el procesamiento de streaming. Estas tareas se encuentran dentro de las que permite llevar a cabo un SPE.

2.1. Objetivos de este trabajo

El objetivo principal de este trabajo es mejorar la performance de la caché de resultados en motores de búsqueda con políticas de admisión, basadas en el uso de modelos de predicción que capturen los cambios de concepto en el tiempo, manteniendo un modelo siempre ajustado. De este objetivo se desprenden objetivos secundarios:

- a. Evaluar al algoritmo Hoeffding Adaptive tree frente al registro de consultas de un motor de búsqueda real, ampliamente utilizado por la comunidad científica.
- b. Extraer atributos que describan diferentes aspectos del comportamiento de los usuarios de un motor de búsqueda y determinar su relevancia para predecir futuras ocurrencias de una consulta.
- c. Diseñar y desarrollar un analizador sintáctico que permita representar gráficamente los modelos generados por los árboles de decisión, con el fin de comprender la hipótesis de clasificación que estos contienen.
- d. Determinar la configuración óptima de HAT para la clasificación de consultas, variando criterios de división y métodos de manejo de atributos numéricos.
- e. Analizar la utilización de recursos de HAT, simulando una fluctuación en la tasa de consultas.
- f. Estudiar la capacidad de adaptación de HAT, en un entorno altamente dinámico, introduciendo artificialmente un cambio de concepto abrupto en el flujo de consultas.

- g. Evaluar la herramienta de software libre (Apache Storm), dedicado al manejo de flujos intensivos de datos en tiempo real y de forma distribuida, para suministrar a HAT los ejemplos que le permiten mantener su modelo actualizado.

2.2. Contribuciones

El principal aporte de este trabajo consiste en mostrar la aptitud del árbol de decisión adaptativo, Hoeffding Adaptive Tree para predecir futuras apariciones de consultas a partir de un flujo continuo e infinito, siendo usado para gestionar la admisión a la caché de resultados. Los experimentos aquí expuestos, permiten constatar la superioridad que los mecanismos de adaptación le otorgan a HAT por sobre C4.5 para llevar a cabo la predicción. Adicionalmente, se evalúa el rendimiento de HAT frente a cambios de concepto de abruptos y tasas de arribo variables, con resultados que satisfacen el comportamiento descripto por sus autores. El segundo aporte es el diseño de un nuevo sistema complementario al de caché, sobre una de las plataformas más populares de procesamiento de flujos de datos en tiempo real (Apache Storm), con el que se generan los datos suministrados al árbol de decisión adaptativo, usados por este para mantener actualizado un modelo al concepto vigente en el flujo de consultas.

2.3. Organización

Capítulo 2: Introducción. En este capítulo se introducen los conceptos básicos de las áreas que conciernen a este trabajo. Al inicio, se introducen conceptos y desafíos que prevalecen en los motores de búsqueda Web. Posteriormente, se describe la evolución en el campo de la minería de datos, profundizando sobre minera de streaming de datos con el objetivo de exponer los avances que son utilizados para atacar la optimización de caché de resultados en los motores de búsqueda.

Capítulo 3. Antecedentes y trabajos relacionados: En este capítulo se profundiza sobre los conceptos y acercamientos utilizados en la literatura para la optimización de memoria caché en motores de búsqueda. Posteriormente, se realiza una revisión de tres trabajos que sientan las bases práctico-teóricas para llevar a cabo esta investigación. Finalmente, se describe los árboles de decisión clásicos y su evolución para el tratamiento de streaming. En esta apartado, se describe en profundidad el funcionamiento de Hoeffding Adaptive Tree, justificando las características que lo hacen óptimo para su utilización en entorno de los motores de búsqueda.

Capítulo 4: Trabajo de campo, análisis y resultados. Este capítulo comienza con la descripción de las tareas de campo efectuadas con relación a la exploración y procesamiento del conjunto de datos utilizado para la evaluación de los algoritmos. A continuación, se da una descripción detallada de la solución propuesta para la optimización de la caché de resultados. Finalmente, se exponen los experimentos llevados a cabo para la evaluación de los algoritmos y se analizan los resultados obtenidos.

Capítulo 5: Conclusiones y trabajos futuros. En este apartado se analizan y presentan los resultados diversos experimentos llevados a cabo con HAT y C4.5, mencionando líneas de investigación futuras a partir de nuevos interrogantes que se presentan a partir de la experimentación.

3. Antecedentes y trabajos relacionados

3.1. Políticas de admisión a cache en motores de búsqueda.

3.1.1. Introducción

Desde la perspectiva de un usuario, hay tres características esperables de un motor de búsqueda: un tiempo de respuesta bajo, contar con una extensa colección de documentos web [23] y alta calidad en sus respuestas. Hoy en día, una colección extensa para un motor de búsqueda comercial se encuentra en el orden de petabytes [23]. Aumentando aún más el desafío, el tiempo habitual de respuesta de los motores de búsqueda web se encuentra restringido por debajo de los cien milisegundos [24]. Esto motiva la búsqueda de nuevas técnicas de optimización que permitan incrementar el número de consultas procesadas por unidad de tiempo [25]. Entre esas técnicas se encuentran el procesamiento paralelo y distribuido, la compresión de índices y el *caching*.

El uso de caché es una práctica común a muchos escenarios para lograr la optimización de performance, incluyendo sistemas operativos, bases de datos, servidores proxy, motores de búsqueda y servidores web. La idea esencial en el uso de caché es almacenar en una memoria rápida elementos que puedan llegar ser requeridos en un futuro cercano [26]. Si un elemento es entregado directamente desde la caché, no solo la latencia de la consulta se reduce sino que también se disminuye parte de la carga de trabajo en los servidores. La latencia de consulta se define como la diferencia entre el tiempo de su arribo al sistema y el tiempo requerido en completar su procesamiento [24]. Parte de la reducción de latencia se explica porque en memoria principal el tiempo de lectura/escritura es de varios órdenes de magnitud menor que el de memoria secundaria. Administrar el espacio de memoria principal de forma eficiente deriva, por lo tanto, en la mejora del rendimiento del sistema, mejorando la tasa de resolución de consultas y optimizando el uso del equipamiento. De cierto modo, esto también involucra una reducción de costos dado que mejora el rendimiento del hardware sin una inversión extra.

3.1.2. Arquitectura de los motores de búsqueda

La arquitectura estándar utilizada para dar soporte a las resolución de consultas se ejecuta sobre un grupo de computadoras interconectadas por una red de área local (*LAN*) [27], conocido como *cluster*. Desde hace ya más de 15 años, las supercomputadoras de uso específico pasaron a estar en desuso y las configuraciones con hardware de propósito general son predominantes en

los sistemas de cómputo de alta performance. La flexibilidad que otorgan los cluster es una de las razones que producen la expansión de su utilización. El número de nodos, la capacidad de memoria por nodo, la cantidad procesadores por nodo y la topología de la red que los interconecta son propiedades que son adaptables al problema a resolver [28].

Dos tipos de nodos constituyen a un cluster típico sobre el que corre de un motor de búsqueda, un nodo intermediario (*broker*) cuya tarea consiste en proveer la interfaz de usuario y asignar una parte del trabajo a los nodos de búsqueda (*search nodes*) en base a algún criterio. Estos últimos resuelven las consultas y devuelven el resultado al broker.

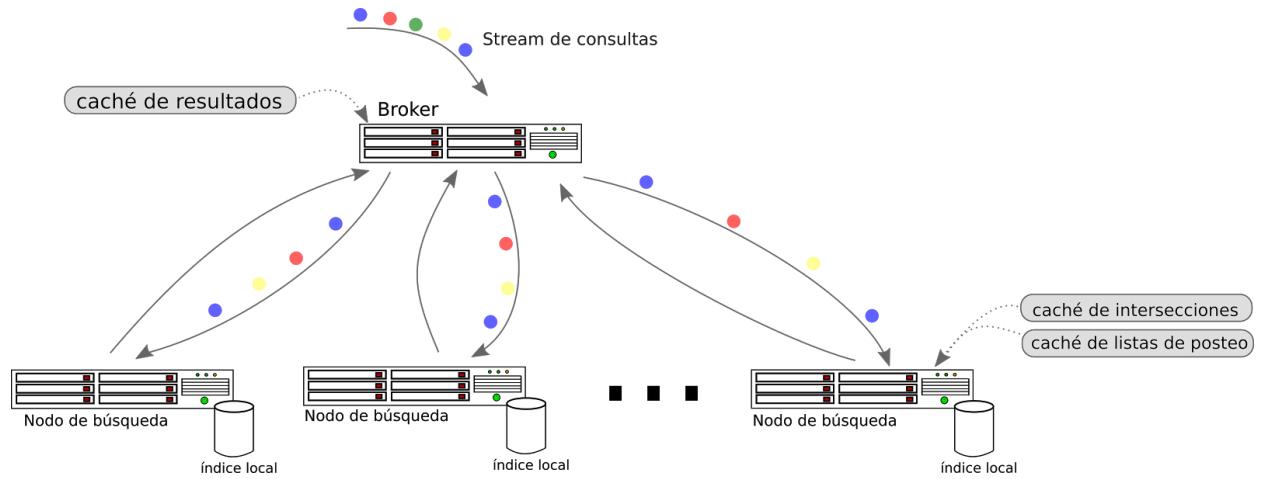


Figura 1 : Representación de la arquitectura estándar de un motor de búsqueda.

El sistema distribuido donde se ejecuta un motor de búsqueda, debe llevar a cabo diversas operaciones a fin de poder responder una consulta al usuario. Cuando el nodo broker recibe una consulta, este primero realiza una búsqueda sobre la caché de resultados. Si la consulta es encontrada, entonces el resultado se entrega directamente. De lo contrario, el broker debe enviar la consulta a los *search nodes*. Estos últimos deben realizar una búsqueda sobre el índice asociadas a los términos de la consulta, procesarla (por ejemplo, descomprimir, calcular similitud entre los documentos y la consulta, etc.) y ordenar los mismos en un ranking en base a algún criterio. Finalmente los nodos de búsqueda retornan al broker los documentos mejor rankeados al usuario [29] . La Figura 1 es la representación del funcionamiento estándar de una arquitectura de un motor de búsqueda, donde generalmente la división de carga de trabajo se realiza fraccionando la colección de documentos en los nodos de búsqueda. Mediante los

círculos se muestra como algunas consultas son respondidas en el nodo broker directamente desde la caché.

3.2. Políticas de caché

En el contexto de los sistemas de búsqueda, disponiendo de una caché con capacidad limitada, las áreas de investigación involucran políticas de admisión a caché, desalojo de caché y *prefetching* [30]. Mientras menor es el tiempo de lectura/escritura que tiene un dispositivo de almacenamiento, mayor suele ser su precio en el mercado. Debido a esto, en la mayoría de los casos, el uso de memoria de acceso rápido (utilizada como caché) es un recurso a optimizar.

Dos tipos de políticas de caché componen la gestión de la misma: admisión y desalojo. Las políticas de admisión determinan qué elementos son aceptados en caché. Cuando la caché se encuentra completa y no queda más espacio para que elementos de mayor utilidad sean almacenados, las políticas de desalojo determinan cuál/es serán removidos para generar lugar.

Es necesario evaluar mediante alguna métrica la performance resultante de la aplicación de las políticas de admisión y desalojo en la caché. Existen alternativas que ponderan distintos aspectos de la resolución de una consulta. Una de ellas, la más usada en la literatura para medir la performance de la caché, es la tasa de aciertos (*hit ratio*) [31]. Este se define como el número de aciertos dividido por el número de consultas [32]. En general, el incremento del hit ratio trae asociado la reducción de carga de trabajo de los servidores y del tiempo de respuesta. Algunos estudios muestran que el costo de procesamiento varía significativamente entre consultas en diversos aspectos: tiempos de procesamiento de CPU, acceso a disco, tiempo de transferencia de red, etc. [33,34]. Por lo tanto, en ciertos casos puede resultar más apropiado el uso de una métrica que tenga en cuenta el costo de procesamiento de las consultas. Por ejemplo, en uno de sus trabajos Feuerstein y Tolosa [35] comparan la performance obtenida con estrategias basadas en costo, frecuencia y la combinación de ambas, empleando el tiempo de ejecución de las consultas.

Las políticas de caché pueden descomponerse en dos tipos: estáticas y dinámicas. La caché estática se llena fuera de línea con consultas realizadas en el pasado, registradas en un query log. La suposición subyacente para usar este acercamiento es que existe un comportamiento estable en la secuencias de consultas del cual el sistema de caché puede sacar ventaja. Dicho de otro modo, las consultas que fueron populares en el pasado, mantendrán una frecuencia alta en el futuro. Por otro lado, en una política de caché de tipo dinámico, la decisión sobre qué consulta se almacena se hace en el momento. Se deben identificar consultas con menor probabilidad de

volver a ser realizadas para que sean removidas, generando espacio para otras. Las políticas dinámicas de caché intentan atacar el comportamiento de ráfaga. Se consideran consultas ráfaga (*bursty queries*), a aquellas que aparecen de forma muy frecuente en un breve periodo de tiempo.

3.2.1. Niveles de jerarquía de caché

La indexación es el proceso de construir un índice sobre una colección de documentos con el objetivo dar soporte a búsquedas eficientes, el índice invertido es la estructura de referencia utilizada en los motores de búsqueda [23]. Una implementación básica consiste en dos estructuras de datos, un vocabulario (*lexicon*) y listas de posteo (*posting list*). La primera de ellas almacena los distintos términos contenidos en los documentos. La segunda es un arreglo de listas que contiene la ocurrencia de los términos dentro de la colección de documentos. Cada elemento de la lista contiene mínimamente el identificador del documento que incluye el término. En relación con la estructura del índice invertido, en la literatura se ha estudiado la utilización de caché en tres niveles de jerarquía [36,37]:

- Caching de resultados: se almacena en caché los docID¹ de los top-k² documentos asociados a una consulta. Se evita buscar las posting list de memoria, procesar las intersecciones y rankear los documentos asociados. La principal ventaja de este método, en relación con la aplicación de aprendizaje de máquina, se encuentra en la cantidad de atributos extraíbles al tratar la consulta como un todo.
- Caching de listas de posteo: se almacenan las listas de posteo relacionadas con los términos de una consulta. Se accede a las listas desde memoria principal y se procesan las intersecciones. Generalmente, se logra un hit rate mayor al del primer método, dado que las listas cacheadas pueden corresponderse con términos de diferentes consultas.
- Caching de intersecciones de listas de posteo: se almacenan en caché las listas de posteo resultantes de la intersección de dos términos. Este método permite reducir los tiempos requeridos para traer las posting list de memoria secundaria y computar la intersección de las mismas.

¹ docID: Identificador único de un documento.

² top-K: K primeros documentos en la lista de resultados.

Habitualmente en el nodo broker se implementa una caché de resultados. En un nivel más alto, resultados asociados a consultas frecuentes son cacheados para ser entregados al usuario sin la necesidad de ser ejecutados una y otra vez [33]. En los search nodes se implementa una caché de listas de posteo y de intersecciones de listas de posteo. Esas listas se corresponden a términos frecuentes que son almacenados en memoria principal para ahorrar tiempos en transferencia desde disco.

3.2.2. Trabajos relacionados

En un trabajo de Tolosa [38] se realiza, entre otros experimentos, el desarrollo de una política de admisión a caché basada en algoritmos de aprendizaje de máquina para mejorar el tiempo total de procesamiento de las consultas. Específicamente, su trabajo respecto a este tema se enfoca en el desarrollo de políticas de admisión de caché dinámicas basadas en la estrategia de resolución S4 [39]. La misma consiste, sintéticamente, en descomponer una consulta en todas sus posibles combinaciones de dos términos. Basándose en un método que desarrollaron en uno de sus trabajos previos, almacenan en caché las listas de posteo de aquellas combinaciones clasificadas como elementos (o intersecciones) frecuentes. Para decidir si un elemento es frecuente utiliza algoritmos aprendizaje supervisado, Naive Bayes y árboles de decisión. El primero de ellos tuvo en sus experimentos bajo desempeño, por lo que decide sustituirlo por otro algoritmo de clasificación. En reemplazo utiliza Random Forest (RF) [40] ensemble, que consiste básicamente en construir diversos árboles de decisión durante la etapa de entrenamiento con el objetivo de reducir la varianza y evitar el sobreajuste. Para entrenar el árbol extraen los siguientes atributos de una intersección:

- TF_t1 y TF_t2: Frecuencia del término 1 y 2, respectivamente, en la colección de documentos.
- DF_t1 y DF_t2: Cantidad de documentos que contienen el término 1 y 2, respectivamente, en la colección de documentos.

Finalmente, los atributos son utilizados por los autores para generar modelos que determinen si la intersección de las listas de posteo de dos términos debe ser almacenada. Mediante el uso de los árboles de decisión el autor obtiene una exactitud en la clasificación de aproximadamente un 75%. Luego realiza un análisis del tipo errores que comete este clasificador, y resalta que es un fallo de mayor gravedad etiquetar a un par de términos como infrecuentes, siendo este en realidad lo contrario. La razón para esto es que podría darse el caso de un ítem que tenga una alta frecuencia de aparición jamás sea cacheado por un error en su clasificación. Los

experimentos de su trabajo fueron realizados con una muestra significativa del query log de AOL [41] con modelos entrenados en modo por lote.

En su trabajo, Tolosa utiliza un modelo Clairvoyant para establecer la performance máxima que puede alcanzar el algoritmo empleado frente a un determinado conjunto de datos. El método supone la existencia de un algoritmo que predice de manera insuperable que elementos volverán a repetirse.

En la sección trabajos futuros, Tolosa [38] propone investigar la dinámica de la frecuencia de cada par de términos para encontrar intervalos óptimos en los cuales resultaría adecuado volver a entrenar un modelo. Inconvenienteamente, usando un algoritmo de árbol de decisión tradicional, es necesario construir el modelo desde cero al requerirse una actualización. La desventaja que presenta este algoritmo da en parte sustento a este trabajo, en el cual se propone propone el uso de un árbol de decisión capaz de mantener actualizado un modelo de predicción, aprendiendo de nuevos ejemplos a medida que llegan y listo para predecir en cualquier momento.

Otro trabajo estrechamente relacionado con esta tesina es el de Kucukyilmaz y otros [42]. Al igual que el anterior trabajo [38], estos investigadores utilizan técnicas de aprendizaje de máquina para mejorar la performance de una caché de resultados, combinando políticas estáticas y dinámicas. En sus experimentos, entrena dos modelos de predicción con árboles de decisión mediante el uso del query log AOL [41], que contiene consultas reales realizadas al motor de búsqueda de la misma empresa durante un periodo de 12 semanas. Realizan un procesamiento del mismo que les permite obtener una enorme diversidad de atributos, usados luego para determinar si una consulta se volverá a ser realizada en el futuro. Entre los atributos con los que construyen los modelos se encuentran: el tipo de consulta, características de los resultados seleccionadas por el usuario (cantidad de resultados seleccionados, cantidad de oportunidades en que el usuario selecciona la primera posición del ranking), frecuencia de las consultas y de sus términos tratados individualmente (en diferentes niveles de agregación basados en tiempo).

Los dos modelos desarrollados por Kucukyilmaz y otros [43] tienen funciones diferentes, aunque relacionadas. Uno de ellos se utiliza para predecir si una consulta es singleton³ y el otro para predecir cuánto tiempo tardará (next Arrival Time - IAT) en volver realizarse la misma. En base al primer valor, se decide si la consulta debe almacenarse (política de admisión). El IAT permite determinar qué ítem debe ser removido de la caché (política de desalojo), dependiendo del espacio requerido serán desalojados aquellos ítems con mayor IAT. En base a sus

³ Consulta que tiene una sola ocurrencia en una secuencia determinada.

experimentos, los investigadores concluyen que no son significativas las mejoras obtenidas utilizando ambos modelos. Esto se debe a que el modelo que clasifica singletons, perteneciente a políticas de admisión, tiene mayor incidencia en la performance final de la caché. En el presente trabajo se desarrolla un modelo para la predicción de singletons, que se corresponde con el tipo de política que, en el trabajo de Kucukyilmaz y otros, se describe como el de mayor efecto en la performance.

Un tercer trabajo relacionado es el de Baeza-Yates y Gionis [27]. En este trabajo analizan el comportamiento de las consultas realizadas a un motor de búsqueda durante el periodo de un año. En el query log que utilizan, la distribución de las consultas cambia lentamente a lo largo del tiempo. En el encuentran que sólo un pequeño porcentaje de las consultas son nuevas. Acorde con lo anterior, también observan que la mayoría de las consultas que aparecen en una cierta semana se repite en las siguientes, durante el periodo de seis meses. Hoeffding Adaptive Tree es a priori apto para ajustar un modelo a este tipo de distribuciones, con cambios graduales en el tiempo.

Los autores realizan experimentos variando la capacidad de caché, que es llenada en combinación de políticas estáticas y dinámicas. El fragmento estático se llena con las consultas de mayor frecuencia colectadas en un periodo de dos semanas, a partir de 128M consultas. La tasa de aciertos se reporta en periodos de una hora durante un lapso de 7 días. En base a esto, observan que el valor más alto es alcanzado durante la noche (cerca 2-3 AM). Los investigadores sugieren que la caché estática resulta efectiva durante una semana completa después de un periodo de entrenamiento. Por lo tanto, el trabajo da la pauta de que un modelo actualizado en periodos inferiores a una semana podría alcanzar una tasa de aciertos superior. En esta tesina, la actualización del modelo de predicción es realizada suministrando a Hoeffding Adaptive Tree nuevas consultas provenientes del pasado, registradas en el query log.

3.3. Árboles de decisión

3.3.1. Algoritmos de árboles de decisión tradicionales

Dentro de los algoritmos supervisados más utilizados de aprendizaje de máquina se encuentran los árboles decisión que usan una estrategia del tipo “divide y vencerás”. Usando esta estrategia los árboles de decisión tienen la facultad de dividir el espacio comprendido por las instancias de

entrenamiento en subespacios y ajustar cada uno de ellos con diferentes modelos [44]. Otro aspecto positivo de los árboles de decisión es su completa expresividad de la hipótesis de clasificación, uno de los motivos de su popularidad [45]. En otras palabras, los árboles de decisión generan modelos que pueden ser analizados fácilmente. Esto permite, por ejemplo, conocer la importancia que tiene un atributo, elegir la profundidad del árbol para que pueda generalizar mejor, comprender valores de división de los atributos y etcétera.

El modelo de clasificación permite direccionar las instancias en base a los valores de sus atributos sobre un árbol compuesto por una raíz, ramas, nodos y hojas. La forma tradicional de representar los elementos descriptos se ejemplifica en la Figura 2. Formalmente, un árbol de decisión es un grafo dirigido acíclico, donde los nodos corresponden atributos y sus aristas representan el valor o rangos de valores de un atributo. El nodo raíz contiene el atributo que mejor separa al conjunto de datos de entrenamiento con respecto a la clase. Ese atributo se escoge en base a una métrica que puede ser, por ejemplo, la ganancia de información o el índice de Gini [46]. Los nodos del grafo pueden ser de decisión o bien, nodos hoja. El primero de ellos define una decisión a tomar frente al valor de un atributo, con dos o más caminos posibles. El segundo tipo, concluye la clase a la que corresponde una instancia. Una instancia es clasificada siguiendo uno de los caminos desde la raíz hasta las hojas, cada camino desde la raíz a las hojas define una conjunción de condiciones con las que es clasificada.

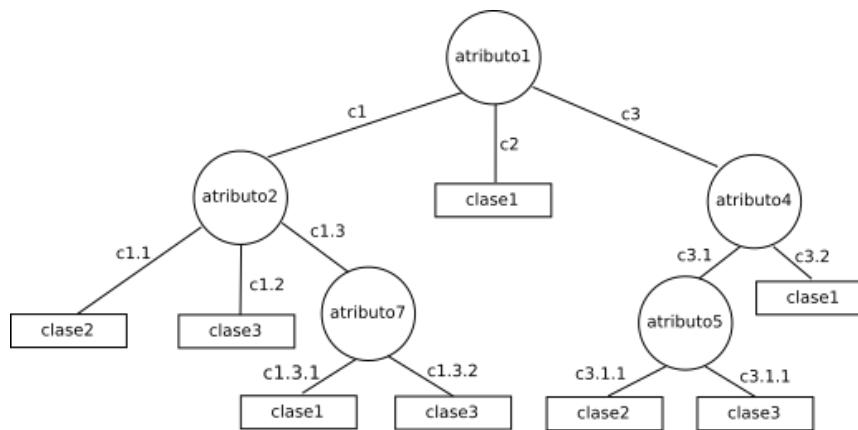


Figura 2: Ejemplo de Árbol de decisión. La letra “C” indica condiciones con las que se clasifican las instancias.

Los tres algoritmos clásicos de árboles de decisión son ID3 (Induction of Decision Tree) [47], CART (Classification And Regression Trees) [48] y C4.5 [49]. ID3, desarrollado por Quinlan en 1986, construye un árbol en base a ganancia de información. Esta es una métrica basada en la entropía de Shannon que intenta medir la cantidad de información provista por un atributo.

C4.5 es la segunda versión desarrollada por Quinlan que incorpora el manejo de atributos continuos y datos faltantes junto con un procedimiento para evitar el sobreajuste de los modelos generados, conocido como poda. El sobreajuste se produce cuando un árbol crece indeterminadamente creando un modelo sensible al ruido y poco robusto [50]. El método que utiliza C4.5 se basa en medir la tasa de error, obtenida con el siguiente cálculo:

$$ErrorRate_n : \frac{E_n}{N_n}$$

Donde:

E: Es el número de instancias que no pertenecen a la clase más frecuente en el nodo n

N: Cantidad de instancias vistas por el nodo n.

A diferencia de los otros dos algoritmos desarrollados por Quinlan, el propuesto por CART escoge qué atributo colocar en cada nodo usando el índice de Gini. Este último algoritmo también posee un método de poda, más complejo que el de los otros dos, que calcula los parámetros usados a través de validación cruzada [50].

Los tres algoritmos mencionados asumen que los ejemplos de los que aprenden se encuentran almacenadas en algún lugar y por lo tanto, realizan lecturas secuenciales de ellos (como mínimo una lectura por cada nivel) para construir el árbol. Por consiguiente, esa estrategia de aprendizaje es impracticable cuando las instancias de entrenamiento provienen de un stream de datos que tiene un tamaño potencialmente infinito. Los árboles tradicionales tienen un cuello de botella en lo que respecta a tiempo y memoria, lo que le impide cuando frente a un conjunto de datos de entrenamiento muy grande hacer uso de todas las instancias disponibles. Como consecuencia, se suelen construir modelos demasiado complejos, incapaces de generalizar cuando se le presentan instancias diferentes a las antes vistas.

Algunos de los algoritmos propuestos antes de Hoeffding Tree, que lograban hacer un uso constante de memoria y tiempo por instancia eran extremadamente sensibles al orden de las instancias de entrenamiento. En otras palabras, si el algoritmo recibe instancias con ruido al inicio de su ejecución puede confinar el modelo a un bajo rendimiento. Por esta razón, en general estos algoritmos alcanzan un rendimiento inferior al que se obtendría con un algoritmo de árboles de decisión tradicional entrenado con un conjunto de instancias muestreado a partir del flujo de datos [51].

3.3.2. Hoeffding Tree

En el año 2000, Domingos y Hulten proponen Hoeffding Tree (cuya implementación lleva el nombre en inglés: Very Fast Decision Tree) [51], un algoritmo que da origen a gran cantidad de variantes y mejoras que constituyen el estado del arte en procesamiento de streaming de datos con árboles de decisión. Este es el primer algoritmo capaz de cumplir con todos los requisitos necesarios para aprender a partir de flujos de datos. Antes de este, los algoritmos más eficientes disponibles extraían mediante muestreo los patrones de las bases de datos con tamaño mayor al de la memoria disponible.

Como se expone en la Sección 3.2.3 los árboles de decisión tradicionales no son capaces de procesar un flujo de datos con tiempo y memoria constante. Hoeffding Tree requiere analizar cada instancias de entrenamiento como máximo una vez, esta es una cualidad fundamental que le permite aprender a partir de flujos de datos que arriban en forma continua. En caso contrario, si el algoritmo careciera esta virtud la cola de espera de instancias que arriban aumentaría indefinidamente su tamaño. El uso de memoria principal se mantiene constante dado que VFDT no requiere almacenar en ella más que la estructura del árbol y los valores estadísticos asociados al mismo. Hoeffding Tree se considera una innovación en el área por ser capaz de construir incrementalmente un árbol de decisión a partir del arribo de nuevas instancias. La cantidad de instancias necesarias en un nodo para hacer crecer sus ramas se obtiene a partir del límite de Hoeffding, también llamado Chernoff aditivo:

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}$$

n: cantidad de instancias vistas

δ : error admitido en la división

R: valor de probabilidad máximo en la distribución de clases

El límite de Hoeffding es independiente de la distribución que genera los datos. Es utilizable en cualquier situación en la cual las observaciones sean independientes y generadas por una distribución estacionaria. Por lo que generalmente resulta más conservador que cuando se usa un

límite específico de una distribución [44], como podría ser una distribución de Poisson o Normal. Esta métrica le da sustento matemático necesario para garantizar, con cierto grado de confianza, que un determinado número de instancias representa a la distribución del stream de datos. En otras palabras, el límite de Hoeffding intenta encontrar el menor número de ejemplos vistos con el que se seleccione el mismo atributo que usando un número infinito de ejemplos. Por lo tanto, esta métrica permite explotar el hecho de que un pequeño número de ejemplos pueden ser a menudo suficientes para determinar el atributo óptimo en la división en una rama. En el inicio el algoritmo utiliza las primeras instancias para elegir el atributo que mejor divide al conjunto de entrenamiento para ubicarlo en la raíz del árbol. Una vez seleccionado el atributo de la raíz, las siguientes instancias son dirigidas desde esta hasta las hojas, los atributos que maximicen la métrica objetivo (Índice de Gini o Ganancia de información) se insertan como nodos del árbol. Este procedimiento es repetido recursivamente para cada nivel del árbol. Aquellas ramas cuyo valor del límite de Hoeffding no alcance el valor de confianza, parámetro ingresado al ejecutar el algoritmo, no son divididas. Esto puede considerarse como un mecanismo de pre-poda de Hoeffding Tree.

Si bien el algoritmo original propuesto por Domingos y Hulten fue sumamente innovador, cuenta con una severa limitación que restringe su uso en un gran número de ambientes. Originalmente VFDT asume que los stream de datos con el que es entrenado es aleatorio y sigue una distribución estacionaria. Sin embargo, en la mayoría de los casos esto no se cumple. Este es el caso de las consultas realizadas a un motor de búsqueda (MDB), donde aparecen algunas que son muy frecuentes por un periodo breve de tiempo, denominadas en la literatura como bursty queries [52]. Un ejemplo de una bursty query es al producirse un terremoto, habrá aumento abrupto de usuarios realizando consultas relacionadas con el tema, que antes eran infrecuentes. Si el algoritmo no es capaz de capturar este cambio para volcarlo en el modelo y por el contrario, continúa clasificando aquellas consultas relacionadas con el tema como infrecuentes, se estará desaprovechando una oportunidad de mejorar la eficiencia de la caché. Esta situación también podría darse a la inversa, suponiendo que existe una consulta frecuente por un periodo de tiempo en el que el modelo la clasifica como tal, cuya frecuencia desciende posteriormente, el algoritmo debería ser capaz de reconocer el cambio para liberar espacio en la caché, necesario para otras consultas con mayor frecuencia.

3.3.2.1. Manejo de atributos numéricos

Infinidad de atributos del mundo real son expresados naturalmente mediante valores numéricos continuos por lo que la capacidad de aprender a partir de los atributos numéricos resulta esencial. Para los algoritmos de minería de datos con funcionamiento por lote el manejo de atributos numéricos es bastante sencillo [53], la razón de esto es que tienen disponible en memoria cada valor numérico. A partir de su disponibilidad, es trivial calcular medidas de dispersión de un atributo (mínimo y máximo), así como, generar una representación mediante rangos que permita determinar la relación de un atributo para con la clase. Los algoritmos con funcionamiento por lote utilizan diferentes técnicas supervisadas y no supervisadas que le permiten segmentar la distribución de los datos.

Lamentablemente, las soluciones para el manejo de atributos numéricos en ambientes de streaming no son tan sencillas debido a que potencialmente todos sus valores podrían ser únicos. Por esta razón, para encontrar los valores de separación óptimos debería contarse con almacenamiento infinito donde mantenerse registro exacto de los valores tomados por el atributo. Los árboles de decisión basados en Hoeffding Tree atacan este problema manteniendo una aproximación de las distribuciones por atributo en las hojas del árbol en las que se espera una división. Diversos autores han propuesto alternativas para el manejo de atributos numéricos, que son descriptas brevemente a continuación.

3.3.2.1.1. VMFL

Este método fue ideado por Domingos y Hulten, los autores del Hoeffding Tree original. Básicamente, consiste en la sumarización de los valores de los atributos mediante la técnica *binning*. Mediante esta técnica aquellos valores de la distribución de datos que caen en un pequeño intervalo (*bins*) son reemplazados por un valor representativo del mismo, habitualmente un valor central. El rango cubierto por cada bin es fijo y no se modifica con más ejemplos vistos. En el inicio por cada valor único que arriba se crea un nuevo bin y específicamente el método VMFL admite un máximo de mil bins. Una vez alcanzando el número máximo de bins, los siguientes valores en el stream actualizan el contador de aquel más cercano. En definitiva, el algoritmo sintetiza la distribución numérica mediante un histograma cuyos bin se establecen con los primeros mil valores únicos recibidos.

Existen dos posibles problemas con este último acercamiento. Uno de ellos se encuentra en que el método es notablemente sensible al orden de los datos. Específicamente, si los primeros mil valores únicos están sesgados hacia un lado del rango de valores, entonces el resumen final de las distribución sería incapaz de representar un nivel de detalle suficiente el rango total de valores. El segundo problema es encontrar la cantidad óptima de bins, puesto que demasiados aumentaran la precisión de la distribución incrementando el costo de almacenamiento. Por el contrario, muy pocos podrían ser insuficientes para encontrar un buen punto de división.

3.3.2.1.2. Árbol binario exhaustivo

Mientras el método VMFL representa de forma aproximada la distribución de un atributo, el método del árbol binario exhaustivo representa exactamente la distribución de los datos a costa de dedicar mayor espacio de almacenamiento. En su ejecución, el algoritmo construye incrementalmente un árbol binario donde los valores que llegan son guardados. Las decisiones de división realizadas con este método resultan idénticas a las que se tomarían al utilizar un método por lotes, como el que usa por ejemplo C4.5. Básicamente, son idénticas porque no se descarta información alguna, a menos que un valor se vuelva a repetir. Además del obstáculo que impone la utilización de almacenamiento de este método también requiere de intensivo procesamiento ya que para realizar una división se prueba con cada punto posible, viéndose la situación agravada cuando el árbol se encuentra desbalanceado.

3.3.2.1.3. Resumen de Quantiles

Existen diferentes métodos de estimación de cuantiles, el método con mejores garantías de exactitud es el de Greenwald y Khanna[54]. Este algoritmo mantiene un conjunto ordenado de tuplas, cada una de las cuales contiene un valor proveniente de un subconjunto del flujo de entrada junto con los límites (mínimo y máximo) implícitos en el rango de valores en cada momento. Luego se define una operación para comprimir la suma de los quantiles, la cual garantiza que el error de sumarización se mantenga por debajo de cierto límite. Puntualmente, para Hoeffding Trees el resumen de quantiles se realiza por clase. Cuando se evalúa en qué puntos hacer la división, cada uno de los valores en las tuplas son probados. Al igual que como sucede con VMFL, el número máximo de tuplas es un parámetro que influye en la calidad de los puntos de división encontrados.

3.3.2.1.4. Aproximación Gaussiana

Este método aproxima la distribución numérica de un atributo por clase usando un espacio constante, a partir de la distribución normal o gaussiana. Se puede representar la distribución incrementalmente almacenando solo una pequeña cantidad de valores en memoria sin padecer de sensibilidad al orden de los datos. Específicamente la distribución es mantenida para cada una de las clases mediante la media, la varianza y el número total de ejemplos.

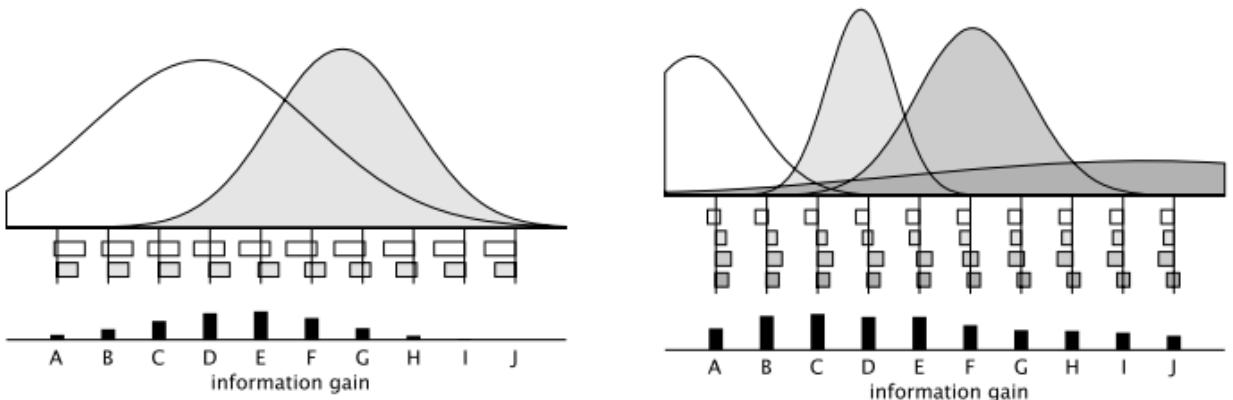


Figura 3: Representación de funcionamiento aproximación Gaussiana de distribuciones de datos con 2 y 4 clases.

La Figura 3 muestra cómo se representan las distribuciones de las diferentes clases de los atributos. Las barras horizontales en ella señalan la división del conjunto de instancias con una determinada partición. Por otro lado, las barras verticales ilustran la ganancia de información en cada división.

Con relación a los métodos de summarización de distribuciones, se debe tener en cuenta que una representación simplificada no resulta necesariamente en desmedro del rendimiento del modelo construido por un árbol. La razón de ello es que pueden existir en diferentes niveles del árbol oportunidades para refinar decisiones de división[53]. Esto puede ser visto en la Sección 5.3. donde los rangos son perfeccionados en más de un nivel del árbol sobre la misma rama para ciertos atributos.

3.3.3. Concept-Adapting very Fast Decision Tree

El desvío o cambio de concepto en el tiempo (time-changing concepts [55] o Concept-Drifting [56]) es la forma en la que se refiere en la literatura la modificación de las propiedades estadísticas de una variable objetivo (que intenta ser predecida por un modelo). En la presencia

de cambios de concepto, el crecimiento incremental no es una propiedad suficiente para evitar la desactualización de un modelo. Los algoritmos de aprendizaje necesitan mecanismos para incorporar los cambios, olvidando datos desactualizados y adaptarse al estado más reciente de los mismos[44]. Concept-Adapting Very Fast Decision Tree (CVFDT) [55] es una extensión de VFDT para adaptarse a cambios de concepto en distribuciones no estacionarias conservando la capacidad de procesar instancias en un tiempo constante. La forma de hacerlo es a través de una ventana deslizante de W instancias, cuando una nueva instancia llega es insertada al principio de la ventana y un número correspondiente de estas es removida al final de la misma. Sin embargo, cada vez que aparece una nueva instancia no se construye un modelo desde cero. En vez de eso, el algoritmo actualiza estadísticas suficientes en cada nodo del árbol incrementando los contadores correspondientes a la nueva instancia y decrementando aquellos a las instancias que son desalojadas de la ventana. Periódicamente, se escanea el Hoeffding Tree y sus árboles alternativos en busca de suficientes estadísticas que sería mejor reemplazar a un determinado árbol por otro. De esta manera CVFDT, genera un árbol alternativo que reemplaza el árbol actual, cuando este último es superado en precisión. Esta segunda versión de Hoeffding tree requiere que se le ingrese como parámetro el tamaño de ventana que debe ser utilizado en la ejecución. Esto último resulta una desventaja, dado que un único tamaño de ventana no será apropiado para cualquier tipo de desvío de concepto. Por lo cual, el valor de W puede ser modificado por el usuario durante su ejecución. En base a sus experimentos, los autores señalan que cuando aparezcan varios nodos de un árbol que sean cuestionables o se produzca un cambio rápido de tasa de arribo, se reduzca el tamaño de la ventana. Por el contrario, en algunas situaciones sería beneficioso un incremento de W cuando haya una mayoría de nodos estables.

El algoritmo construye los árboles alternativos usando tres conjuntos disjuntos de instancias construidos a partir de las instancias que arriban (S_0, S_1, S_2), siguiendo los siguientes pasos:

1. Despues de cada instancia en S_0 , CVFDT recorre todo el árbol, controlando por cada nodo si el atributo haciendo la división sigue siendo el mejor. Si existe un atributo que haga una mejor división, se hace crecer un árbol alternativo con raíz en el nodo, usando las estadísticas contenidas en este.
2. Una vez que el árbol alternativo es creado, las siguientes S_1 instancias son usadas para construir el árbol alternativo.

- Después del arribo de las instancias S_1 , las siguientes S_2 instancias son usadas para probar la exactitud del árbol alternativo. Si el árbol alternativo es más exacto que el árbol vigente en ese nodo, CVFDT reemplaza al árbol vigente por el alternativo.

Los tamaños por defecto que tienen los conjuntos son $S_0 = 10.000$, $S_1 = 9.000$ y $S_2 = 1.000$.

3.3.4. Hoeffding Adaptive Tree

Elegir un tamaño de ventana estático representa tener preconceptos de que tan rápido o cuán a menudo los datos evolucionan. Por lo tanto, quizás ningún valor de ventana fijo sea el correcto, dado que un stream de datos puede tener una combinación de cambios abruptos, graduales y largos períodos de estacionalidad. Hoeffding Adaptive Tree (HAT) [57] es una variante de Hoeffding Tree que utiliza ventanas deslizantes para mantener ajustado el árbol, sin embargo no requiere que el usuario le especifique el tamaño de ventana a utilizar. Esto se debe a que el tamaño de ventana óptimo se calcula individualmente para cada nodo, utilizando detectores de cambios y estimadores llamados ADWIN [58].

El cambio de concepto en las distribuciones de datos a través del tiempo es uno de los principales problemas de la minería de datos y aprendizaje de máquina. Para aprender de esos datos, se necesitan estrategias dirigidas a llevar a cabo las siguientes tareas [59]:

- Detectar cuando ocurre un cambio de concepto
- Decidir qué ejemplos mantener y cuales olvidar.
- Revisar el modelo actual cuando cambios significativos hayan sido detectados.

La idea central en HAT es encapsular en contadores y acumuladores todos los cálculos estadísticos que permiten detectar cambios en la distribución de los datos. Una instancia de ADWIN es colocada por cada nodo del árbol, por decidir individualmente cuantas instancias resultan útiles en cada rama del árbol. De esta forma, ADWIN automáticamente hace crecer el tamaño de ventana cuando no hay ningún desvío aparente y reduce la misma cuando se detecta alguno. El algoritmo intenta encontrar un tamaño óptimo que tenga un balance entre tiempo de reacción y una alta sensibilidad a pequeñas varianzas [59]. HAT implementa la segunda versión del algoritmo ADWIN, en la cual se introducen diversas mejoras en cuanto a uso eficiente de recursos. La primera versión de ADWIN es computacionalmente costosa, dado que controla exhaustivamente posibles divisiones de la ventana actual y mantiene las instancias de forma explícita. A diferencia de la primera versión, la segunda utiliza una variación de histograma

exponencial [60] que le permite comprimir los datos con una cota superior asintótica de $O(\log W)$ en uso de memoria y tiempo de procesamiento.

Vinculado al uso de ADWIN, existe una diferencia importante en el mecanismo de actualización del modelo entre HAT y CVFDT. HAT crea árboles alternativos tan pronto como se detecta un cambio, sin la necesidad de esperar a tener un número prefijado de instancias. De igual manera, apenas existe evidencia de un árbol alternativo alcanza mayor exactitud, se reemplaza el árbol actual. Mientras más abrupto es el cambio, más rápido será creado un árbol alternativo y reemplazado el árbol desactualizado.

3.4. Stream Processing Engines

3.4.1. Introducción

En general, el uso de código específico es utilizado para resolver problemas de baja latencia y gran volumen. Normalmente este tipo de desarrollo se realiza utilizando lenguajes de programación cercanos al bajo nivel. El problema con esto es que las implementaciones de este tipo tienen un gran costo de desarrollo y mantenimiento [61]. Un ejemplo clásico lo constituye un programa desarrollado en C utilizando MPI⁴, cuyo overhead de procesamiento es reducido pero su costo de implementación es alto si se lo compara, por ejemplo, con el requerido para desarrollar un programa en java usando un Stream Processing Engine (SPE).

Un SPE es un framework que tiene por objetivo abordar el desafío de procesar grandes volúmenes de datos, en tiempo real y sin requerir el uso de código específico. Sobre los SPE es posible implementar algoritmos de machine learning para extraer conocimiento de los stream de datos.

3.4.2. Apache Storm

Dentro del ecosistema de aplicaciones de Hadoop, Apache ofrece Storm, un SPE que puede manejar la complejidad de bajo nivel de un sistema distribuido, realizando planificación de tareas, manejo de errores y comunicación entre procesos [62]. Esta característica permite escalar en número de equipos que procesan juntos un stream de datos, abstrayendo al usuario de manejo paso de mensajes y sincronización entre procesos.

⁴ <https://www.open-mpi.org/>

Apache Storm tiene algunos parecidos con Apache Hadoop, aunque su diferencia principal radica en que este último realiza procesamiento por lotes, Storm lo hace en modo por instancias.

3.4.2.1. Elementos principales de Apache Storm

La lógica de la comunicación y coordinación de tareas del sistema distribuido sobre el que ejecuta Apache Storm se encapsula sobre una topología, estructura que por defecto se ejecuta indefinidamente hasta que se le indique lo contrario. Básicamente, una topología puede ser representada como se muestra en la Figura 4 y se compone de dos tipos de abstracciones, llamadas Spouts y los Bolts.

- Spouts: son la entrada de un stream de datos en la topología. Generalmente su función consistirá en leer tuplas de datos desde una fuente externa e ingresarlas dentro de la topología. Un ejemplo de su función es la lectura tweets desde la API de Twitter. Otro ejemplo, más relacionado con el presente trabajo, es la lectura de registros de una base de datos que mantiene la frecuencia de aparición de consultas. Los Spouts pueden ser de tipo seguro o inseguro. Al producirse un error en el procesamiento de una tupla dentro la topología Storm, los Spouts seguros tienen la capacidad de emitirla nuevamente. A diferencia de los anteriores, los Spouts de tipo inseguro no realizan seguimiento de las tuplas por lo que al producirse un fallo no realizan acción alguna.
- Bolts: realizan cualquier tarea de procesamiento requerida en la topología. Esas tareas en general están vinculadas al filtrado, summarización, union, etc. Una vez más, un ejemplo relacionado el presente trabajo es la actualización de la frecuencia de aparición de los términos durante la último minuto, hora o dia.

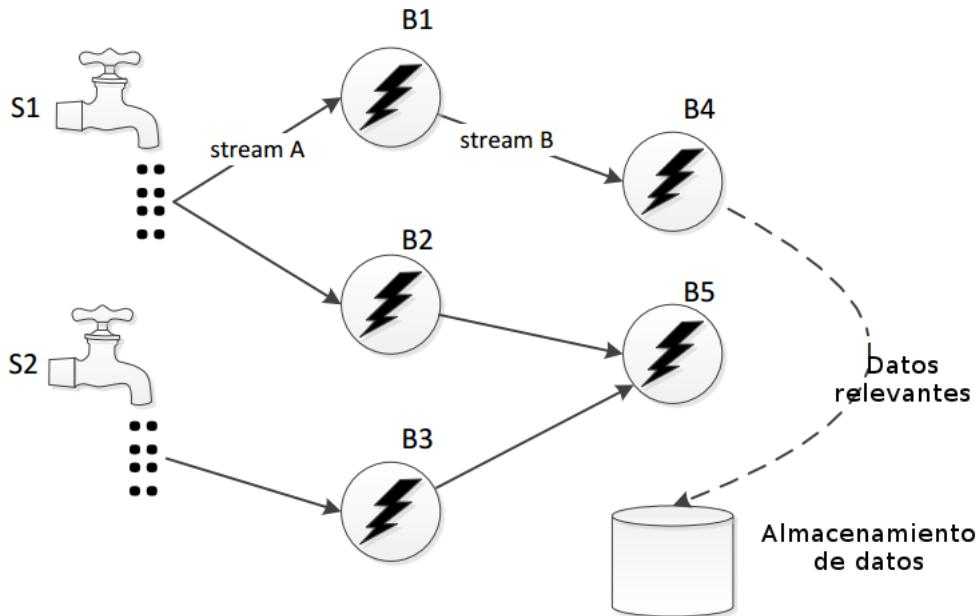


Figura 4 : Ejemplo de una topología de Apache Storm. Los identificadores que comienzan con S corresponden a Spouts y los B corresponde a Bolts.

4. Metodología y propuesta

4.1. Preprocesamiento y Análisis exploratorio del conjunto de datos

4.1.1. Introducción

En este apartado se realiza una breve descripción del conjunto de datos (registro de consultas) utilizados para evaluación de los algoritmos. Luego, se describe el preprocesamiento realizado, con el objetivo de posibilitar la replicabilidad de los experimentos. Finalmente, se realiza un análisis exploratorio del registro de consultas, post procesamiento, para analizar la relación de los atributos con la clase objetivo y evitar la propagación de posibles errores en las tareas realizadas.

Los experimentos en este trabajo son realizados utilizando un registro de consultas (*query log*) de un motor de búsqueda real publicado por la compañía AOL[63] para el uso de la comunidad científica. El mismo contiene aproximadamente 20 millones de consultas, realizadas por 650 mil usuarios durante el periodo de 3 meses. En el mismo, las consultas se encuentran ordenadas

en base a dos atributos: identificador de usuario (AnonID), luego temporalmente. La colección se encuentra almacenada en 10 archivos de texto que poseen los siguientes campos:

- **AnonID:** número de usuario anónimo.
- **Query:** consulta realizada por el usuario, en minúscula.
- **QueryTime:** horario en que la consulta fue realizada. Ej: 2006-03-22 21:38:25
- **ItemRank:** Ranking del resultado seleccionado por el usuario
- **ClickURL:** Dominio de la URL seleccionada por el usuario. Este campo se encontrara presente cuando el usuario seleccionó una URL entre los resultados.

Las líneas en el dataset pueden ser de dos tipos:

1. El usuario realiza la consulta y no hace click sobre ningún ítem de la lista de resultados.

Ejemplo: **479 also sprach zarathustra 2006-03-02 14:48:55**

2. El usuario realiza una consulta. A continuación, selecciona un ítem de la lista de resultados.

Por cada ítem seleccionado por el usuario el query log contiene una entrada que registra la posición del ítem seleccionado y el dominio asociado al mismo.

Ejemplo: **479 family guy movie references 2006-03-03 22:37:46 1 http://www.familyguyfiles.com**

El query log es procesado para extraer características que permitan generar un modelo con el cual clasificar consultas como frecuentes o infrecuentes. Las características obtenidas, son presentadas y analizadas en detalle en la Sección 3.2.

4.1.2. Preprocesamiento

Entre los atributos de mayor importancia para el modelo se encuentran los relacionados con la frecuencia de los términos y de consultas. Para su extracción, ya sea en modo por lote o en línea, se requiere mantener contadores con tiempo de expiración. Esta tarea demanda un uso significativo de procesamiento y espacio de memoria. Puntualmente, utilizando la técnica más inocente y directa, el espacio en memoria requerido para mantener los contadores es superior al disponible. En esta solución, se recorre el conjunto de datos acumulando la frecuencia por minutos, horas y días en dos niveles, primero por términos contenidos en la consulta y luego la consulta completa. La tarea es llevada a cabo utilizando una estructura de datos que permita acumular las frecuencias de niveles de agregación. Para ello, se utiliza un diccionario del lenguaje Python. En la Tabla 1 se muestra como se obtienen las frecuencias asociadas a los dos niveles.

Tabla 1 : Ejemplo de la representación por unidad de tiempo de términos y consultas. El ejemplo 1 corresponde a la cantidad de apariciones del término “tire” en el dia 2006-03-01. Mientras que el ejemplo 2 se corresponde con la cantidad de apariciones de la consulta “how to change a flat tire” en el mismo día a las 14hs.

Ejemplo	Id tiempo	Término/consulta	Cantidad
1	01032006	tire	5
2	0103200614	how to change a flat tire	1

El preprocessamiento tiene como resultado la generación de un archivo intermedio que permite asociar a cada consulta los siguientes atributos:

- Frecuencia mínima de los términos en el minuto/hora/dia que fue realizada.
- Frecuencia máxima de los términos en el minuto/hora/dia que fue realizada.
- Frecuencia promedio de los términos en el minuto/hora/dia que fue realizada.
- Frecuencia mínima de la consulta en el minuto/hora/dia que fue realizada.
- Frecuencia máxima de la consulta en el minuto/hora/dia que fue realizada.
- Frecuencia promedio de la consulta en el minuto/hora/dia que fue realizada.

Además de realizar individualmente el cálculo de frecuencias de términos y consultas, se segmenta en intervalos de una semana las frecuencias para reducir el espacio utilizado. Una vez finalizado el cálculo parcial de una semana, se almacenan en el disco las frecuencias en archivos diferentes por cada día. Esto permite reducir los tiempos de búsqueda en la siguiente fase del preprocessamiento. En particular, el procedimiento para asociar consultas y términos con su frecuencias de aparición, consiste en:

1. Leer consulta (Ej: 121 casa perro 2016-03-03 10:54:22)
2. Dar formato a la fecha en la que fue realizada (Ej: 2006-03-03 → 20060303).
3. Abrir archivo con nombre “03032006”
4. Buscar la frecuencia asociada a la consultas y a sus términos, por cada unidad de tiempo.
Por ejemplo, para buscar la frecuencia del término “perro” en el minuto asociado, buscamos la cadena de caracteres “200603031054”, seguida por el término.

4.1.2.1. Datos faltantes

Un 2,74% de las 20 millones de consultas que contiene el query log de AOL no poseen términos. Ante esta situación, un motor de búsqueda podría retornar un mensaje de error, publicidad o algún resultado popular. En consecuencia, predecir si una consulta sin términos volverá a realizarse carece de sentido, puesto que no se obtiene ningún beneficio con respecto al ahorro de recursos o reducción de tiempos de procesamiento. Por este motivo, son removidas del conjunto de datos las consultas sin términos. Una vez eliminadas, se procede a asignar un identificador a las consultas restantes, lo que resulta de utilidad para realizar operaciones de unión en la etapa de transformación de los atributos.

4.1.2.2. Detección de palabras con errores de escritura

Los motores de búsqueda toman como entrada el texto que los usuarios escriben en el campo de búsqueda, se procesa esa entrada y como respuesta se presenta una lista ordenada de resultados asociados a la cadena de caracteres ingresada. Uno de los factores más importantes que derivan en la obtención de resultados irrelevantes para los usuarios son términos con errores de escritura dentro de las consultas [64]. Aproximadamente entre un 10% y 15% de las consultas realizadas a un motor de búsqueda poseen términos con errores de escritura[65]. Carece de sentido almacenar resultados asociados a consultas que poseen términos de tal tipo debido a la amplia diversidad de errores que puede existir. La diversidad produce que las consultas con errores de escritura tengan frecuencia baja o nula. Por otro lado, la detección de términos mal escritos resulta una tarea con cierta inexactitud y eliminar esas consultas puede ser imprudente. Como alternativa, se incorpora un atributo que indica si se encontraron términos con errores de escritura y el modelo generado determinará si debe almacenarse en caché la consulta.

El mecanismo principal utilizado para detectar palabras con errores de escritura es HunspellBrid, un corrector ortográfico desarrollado en el lenguaje de programación Java. Esta librería puede ser utilizada con cualquier idioma, siendo instanciada con el diccionario correspondiente. A partir de un breve análisis de los términos de las consultas se detectó que la mayoría de ellos corresponden a los idiomas inglés y español, siendo esta configurada para que identifique términos conocidos de ambos idiomas. Antes de procesar un término mediante la librería es necesario filtrar aquellos que correspondan a:

- URL
- Nombres de compañías
- Nombres propios
- Remover consultas sin términos

El siguiente pseudocódigo resume el conjunto de pruebas realizadas a una consulta con el objetivo de determinar si posee términos con errores de ortografía:

```

errorEscritura = False
para cada query in queryLog hacer
    para cada termino in query hacer
        si largoTermino(termino) < 25 entonces
            | errorEscritura = True
        en otro caso
            | si not esTerminoPropio(termino) and not esUnaMarca(termino) entonces
                |   | si not enDicEspañol(termino) and not enDicIngles(termino) entonces
                    |       | errorEscritura = True
            fin
        fin
    fin

```

Algoritmo 1: Pseudocódigo - Detección de errores de escritura

4.1.3. Exploración de los datos

El análisis exploratorio de los datos es un tarea que provee herramientas conceptuales y computacionales para descubrir patrones que promuevan el desarrollo y refinamiento de una hipótesis [66]. En esta sección se explora el comportamiento de los atributos generados para el registro de consultas de AOL con dos objetivos principales. Uno de ellos, entender cómo se relacionan los atributos con la variable objetivo, es decir el atributo que tiene valor ‘singleton’ o ‘no singleton’. En la etapa de preprocesamiento, se llevan a cabo diversas operaciones en las cuales existe la posibilidad de cometer errores. Debido a esto último, el segundo objetivo del análisis exploratorio de datos es evitar propagación de errores a la fase de entrenamiento y evaluación, los cuales seguramente generen una idea inexacta del rendimiento de clasificación de los modelos construidos por los algoritmos de aprendizaje.

El criterio usado para asignar la clase a las consultas es altamente influyente en la evaluación de los árboles de decisión empleados para determinar si estas deben ser admitida en caché. Cuando

una consulta llega por primera vez a un motor de búsqueda, resulta naturalmente imposible que los resultados asociados a esta se encuentren almacenados. En la literatura esta situación es denominada fallo obligatorio (*compulsory miss*) [67] y se intenta reproducir la misma en la forma en que se asigna la clase a las consultas. Para ello, el registro de consultas se recorre secuencialmente y a partir de la segunda aparición de una consulta esta es etiquetada como ‘no singleton’.

El objetivo de este primer análisis es determinar el porcentaje de consultas que aparecen más de una vez en todo el conjunto de datos. Los archivos resultantes del procesamiento descripto en la Sección 4.1.2. se obtienen usando una estructura de datos que permite contabilizar la aparición de las consultas. La asignación de la clase a las consultas resulta en un 58,48% de frecuentes, como se expone en la Figura 5. Este último porcentaje establece el límite superior del rendimiento de la caché de resultados.

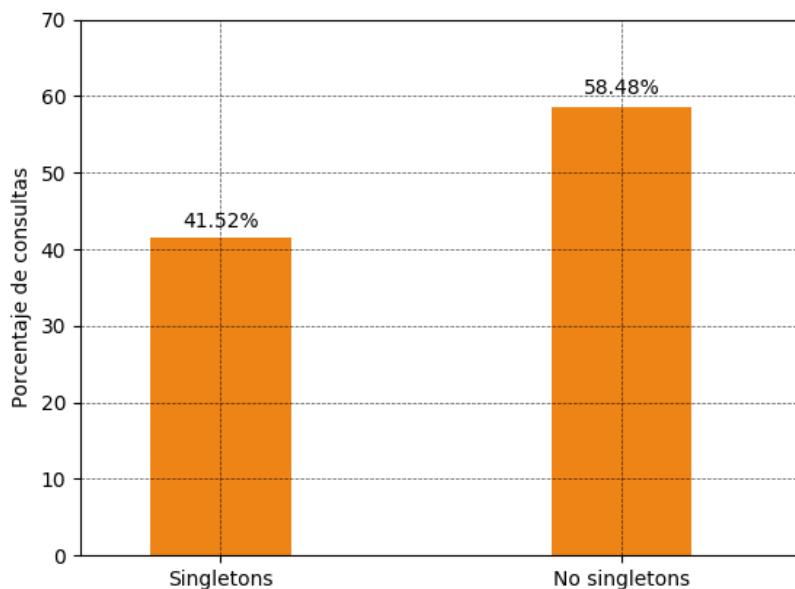


Figura 5: Porcentaje de consultas clasificadas como Singleton y No Singleton

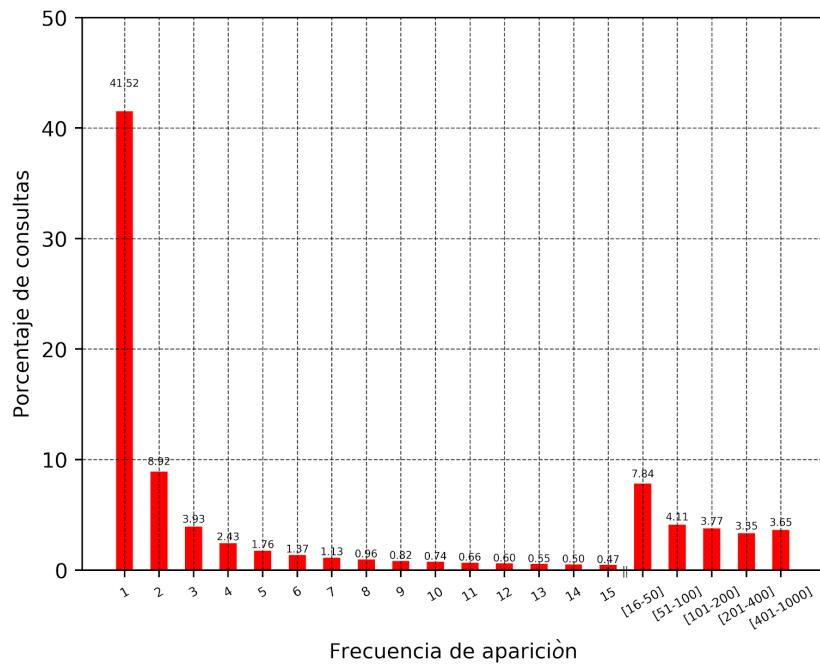


Figura 6 :Distribución de frecuencias de aparición de las consultas del dataset

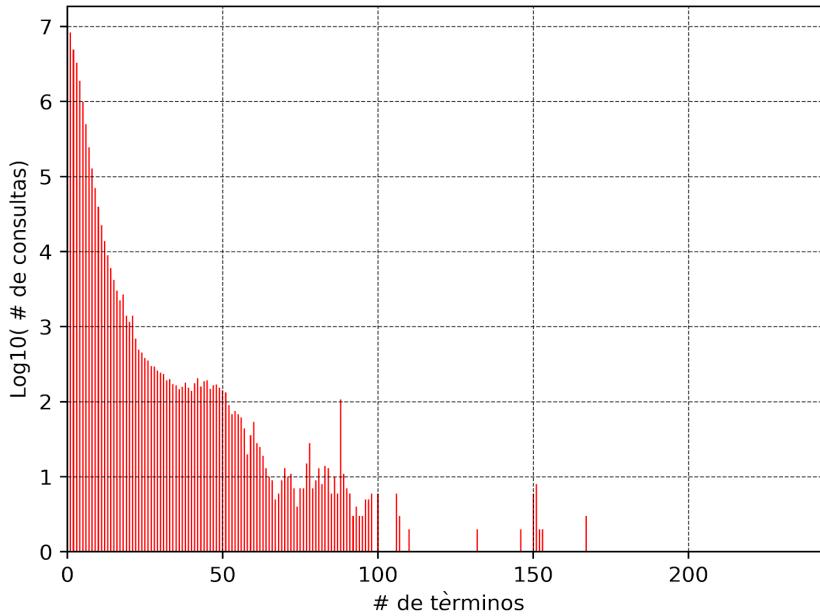


Figura 7 :Distribución de frecuencias de aparición de las consultas del query log con relación a la cantidad de términos. Se grafica la frecuencia de las consultas en escala logarítmica para mejorar la visualización de valores bajos.

Aproximadamente un 22,72% de las consultas tiene una frecuencia de aparición mayor a 16. Este valor da una idea de la mejora obtenible al almacenar en caché consultas con alta frecuencia de aparición. En la Figura 6, un gráfico de barras ilustra la distribución de frecuencias de aparición.

En el query log de AOL, las consultas con menor cantidad de términos poseen una mayor frecuencia de aparición, esta tendencia se expone también en otros trabajos[68,69] (ver Figura 7 y Figura 8). Por esta razón, la cantidad de términos de una consulta es un atributo que se espera que aporte positivamente al modelo de clasificación generado por los árboles de decisión.

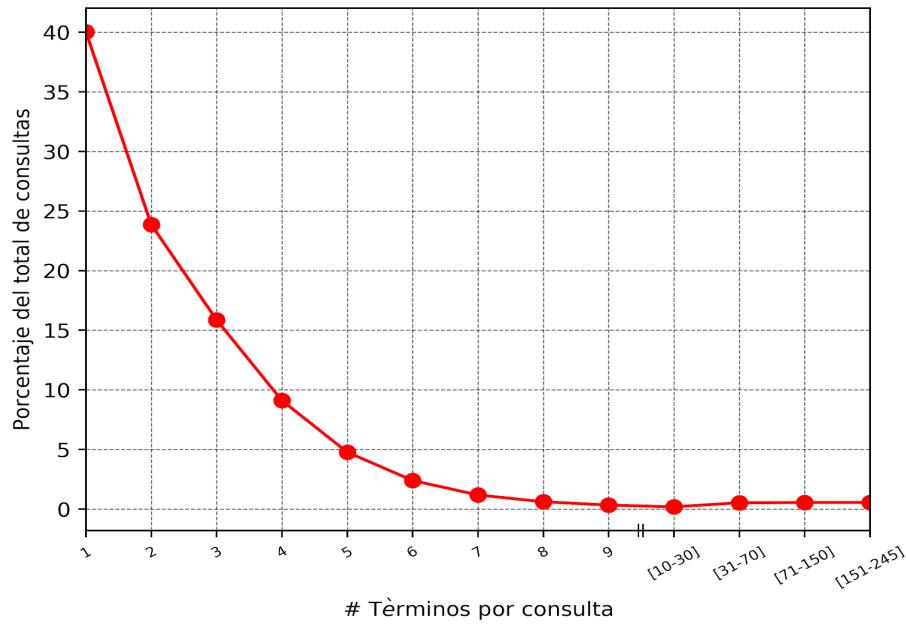


Figura 8 : Porcentaje de consultas con relación a la cantidad de términos

Entre las consultas clasificadas como no singleton, una gran mayoría posee menos de tres términos, comportamiento visualizable en la Figura 9. Puntualmente, ese subconjunto representa aproximadamente el 65% del total de consultas. Otra particularidad que se presenta es que predominan para consultas con un número de términos mayor a tres aquellas que tienen una única aparición.

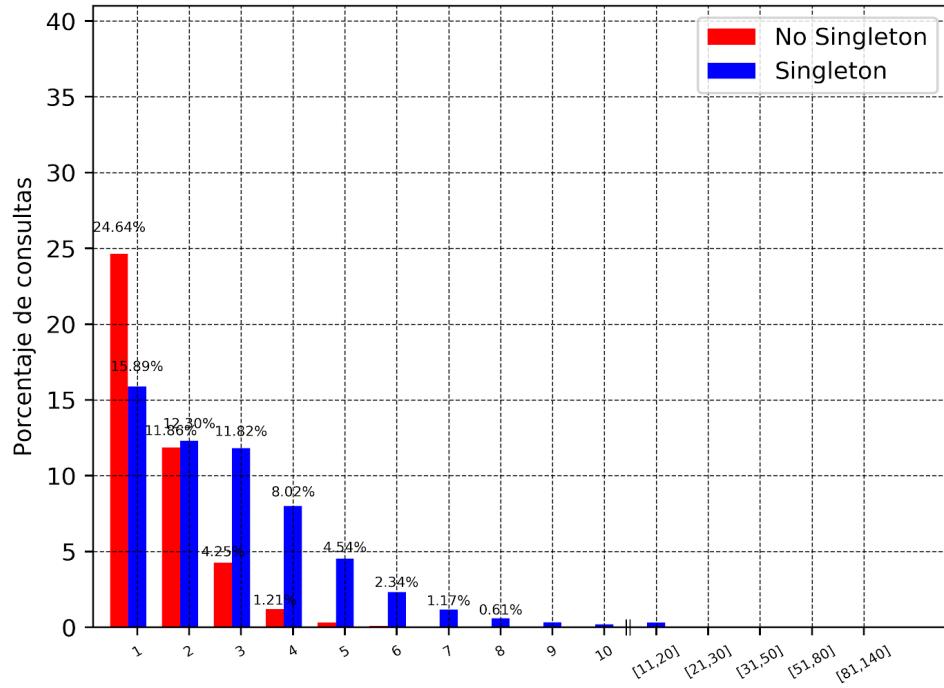


Figura 9 : Relación entre la cantidad de términos y la frecuencia de aparición.

La proporción de consultas frecuentes e infrecuentes es similar entre aquellas con diferentes largos promedio de términos, como lo muestra la Figura 10. En efecto, el largo promedio de los términos, para este conjunto de datos, no será demasiado influyente para la clasificación.

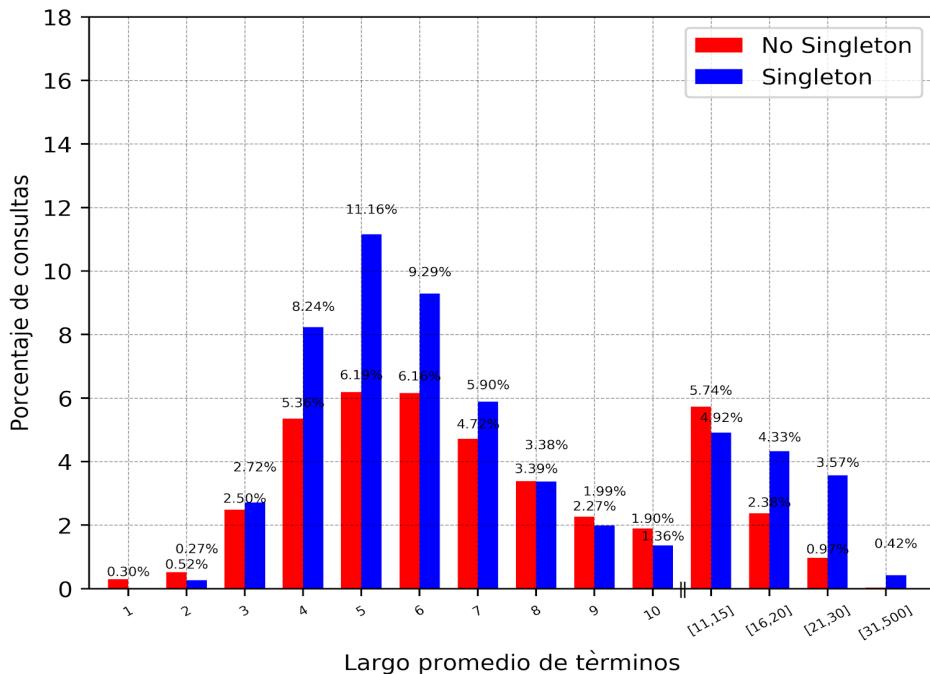


Figura 10 : Relación entre el largo promedio de términos y la frecuencia de aparición de las consultas.

La detección de errores de escritura es un desafío importante por tres principales razones. La primera de ellas, los errores en las consultas realizadas a los motores de búsqueda de tipo web son muy habituales. En segundo lugar, la mayoría de las consultas que realizan consisten en unas pocas palabras clave o términos y no en sentencias gramaticales, haciendo que resulte inapropiado aplicar un acercamiento basado en gramática. La tercera razón, quizás la que más dificulta la tarea, muchas consultas contienen sustantivos y nombres entre sus términos los cuales no se encuentran establecidos en el lenguaje [70]. El procedimiento llevado a cabo la detección de errores de escritura se describe en la Sección 4.1.2.2. Como intuitivamente es esperable, en el query log se detectan más errores de escritura en aquellas consultas que aparecen una única vez. Los distribución entre consultas que poseen estos errores con relación a su frecuencia se expone en la Figura 11.

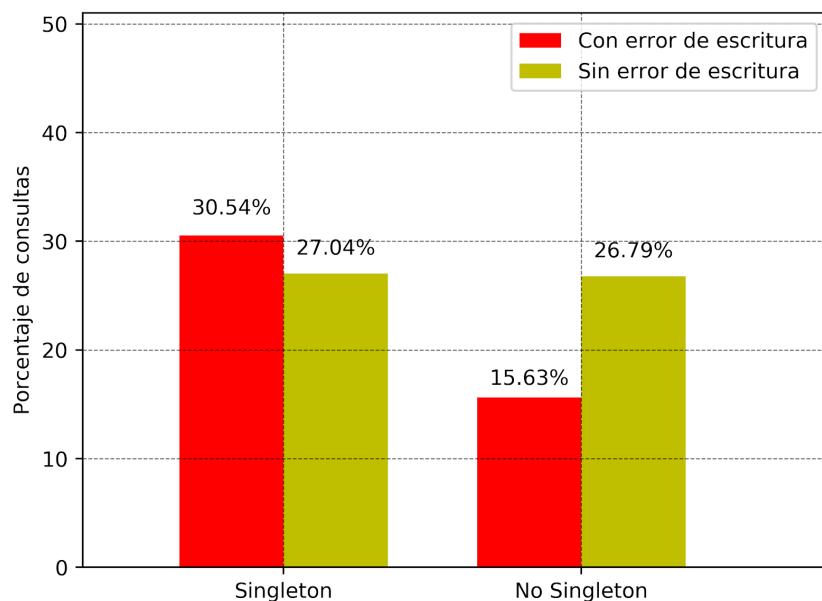


Figura 11 : Relación entre las clase objetivo y errores de escritura detectados en los términos de la consulta

Con el objetivo de determinar si entre los términos de sus consultas los usuarios incluyeron dominios de internet y si esto influye de alguna manera en la frecuencia de aparición de las consultas se usan expresiones regulares para reconocer patrones asociados a los dominios. El resultado de esta tarea se expone en la Figura 12. Las consultas donde se detectan dominios representan un 23% considerando ambas clases. Como resultado de ese análisis, se concluye que la proporción de consultas que poseen dominios es similar en ambas clases.

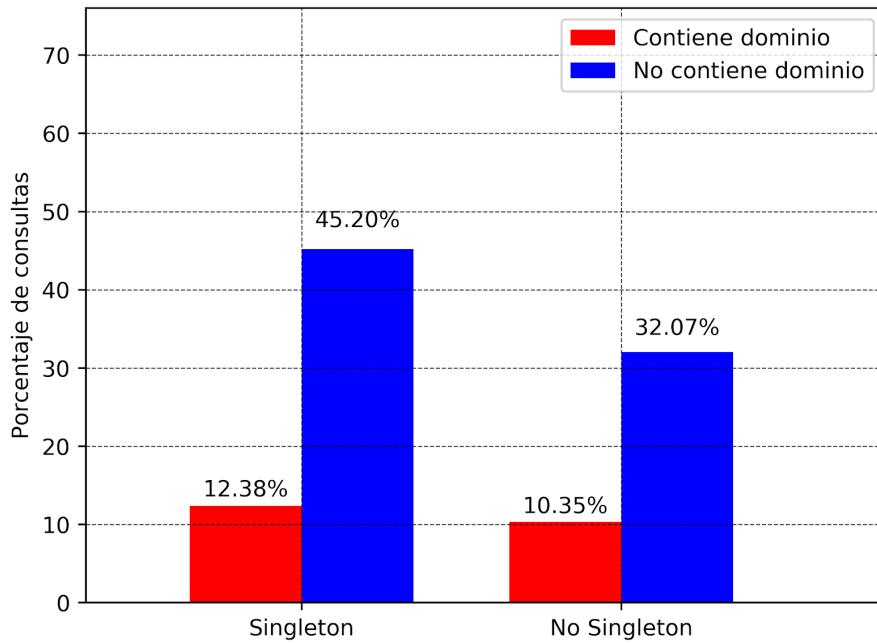


Figura 12 : Relación entre la clase objetivo y consultas en las cuales se encontraron dominios de internet entre sus términos.

4.1.3.1. Sesiones de búsqueda

Una sesión de búsqueda es una serie de consultas realizadas por un usuario en un pequeño periodo de tiempo [68]. Se agrupa las diferentes sesiones de búsqueda de los usuarios utilizando el atributo que registra fecha y hora de realización de la consulta, ambos presentes en el query log. Específicamente, el criterio usado consiste en separar aquellas consultas pertenecientes a un mismo usuario, cuya diferencia de tiempo supere los 40 minutos.

4.1.3.1.1. Cantidad de consultas realizadas

La cantidad de consultas es el número de ellas que realiza un usuario durante una sesión de búsqueda. La media de la distribución es 354, sin embargo existen algunos usuarios que llegaron a hacer más de 40.000 consultas en una sesión de búsqueda, análisis expuesto en la Figura 13. El último valor es llamativo, ya que resulta imposible que un humano realice tal número de consultas en un lapso de tiempo tan acotado. Esto hace suponer que las mismas fueron realizadas a través de algún *metasearch engine*, un motor de búsqueda que utiliza otros para producir sus propios resultados (Ejemplo: startpage⁵).

⁵ <https://www.startpage.com/>

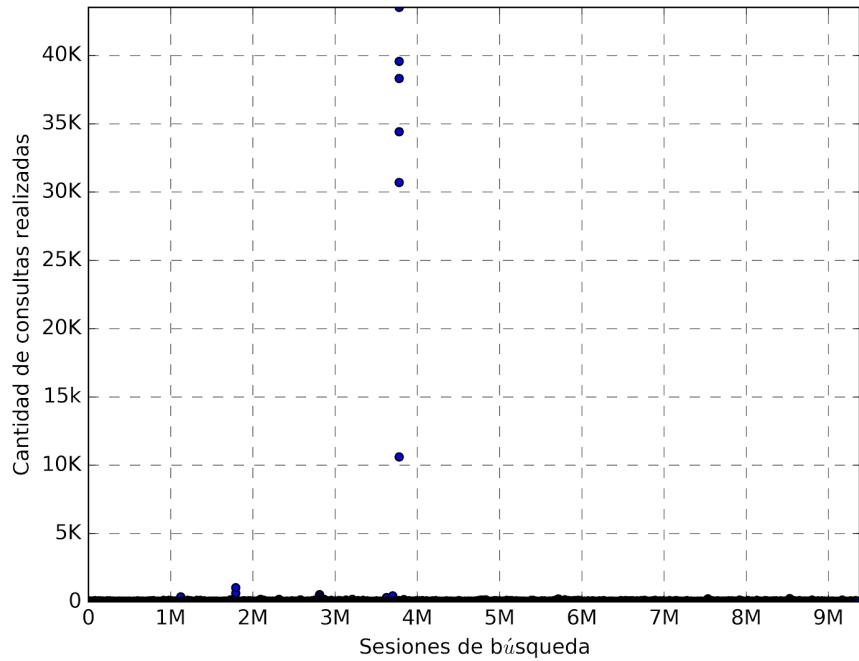


Figura 13 : Cantidad de consultas realizadas por los usuarios

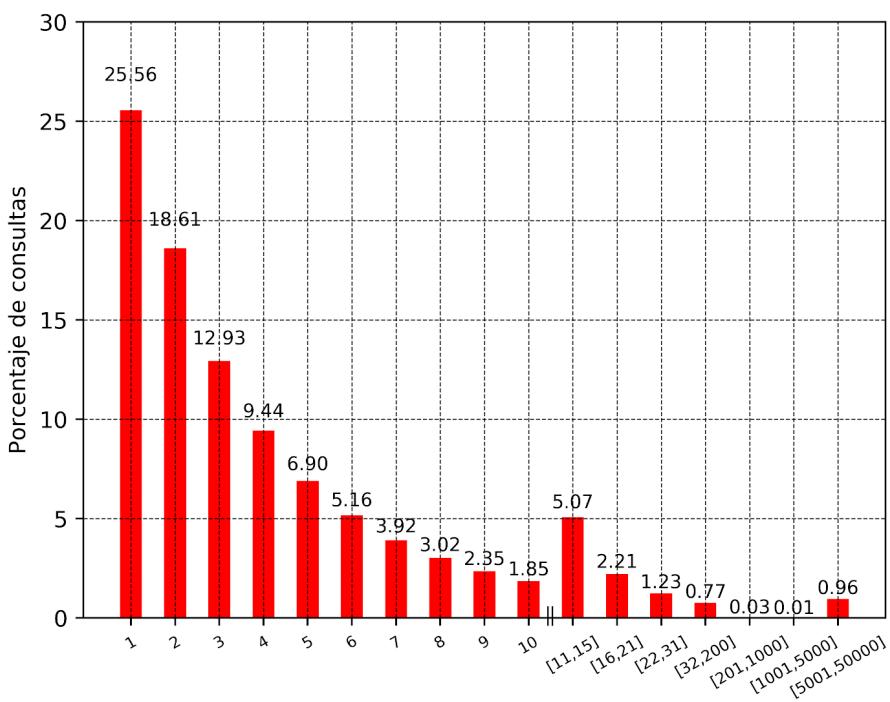


Figura 14 : Porcentaje de consultas en función de la cantidad de ellas realizadas en la sesión de búsqueda.

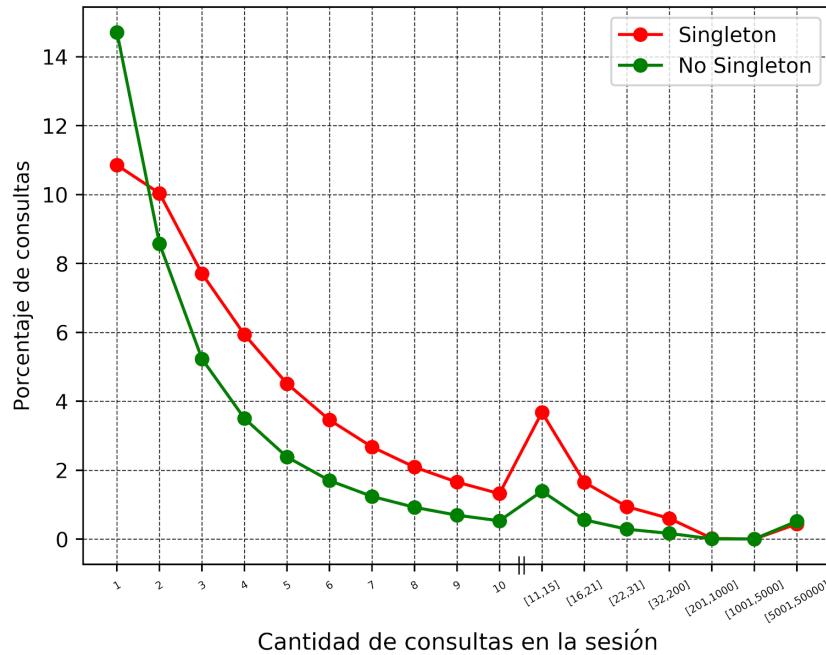


Figura 15 : Relación entre la clase objetivo y la cantidad consultas realizadas.

Las sesiones de búsqueda con menos de cuatro consultas superan la mitad de todas las consultas del conjunto de datos (aproximadamente un 57%), comportamiento apreciable en la Figura 14. Con relación a su frecuencia, las sesiones de búsqueda más cortas, con menos de 2 consultas, son en mayor proporción no singleton (representado en la Figura 15) . Se puede interpretar a partir de esto último que las sesiones de búsqueda acotadas tienden a corresponderse con consultas populares.

4.1.3.1.2. Cantidad de resultados seleccionados en la sesión

La cantidad de resultados seleccionados se obtiene contando por cada sesión de búsqueda la cantidad de dominios en los que hizo el click el usuario. En este caso, la media de la distribución del atributo ‘cantidad de resultados seleccionados’ es 171,4. Un valor que es apenas inferior a la mitad del promedio del atributo ‘cantidad de consultas realizadas’; dos atributos que están íntimamente relacionados. Apoyando esta afirmación, en la Figura 16 se puede apreciar que el valor máximo de resultados seleccionados, al igual que como sucede con la media, es aproximadamente la mitad del valor máximo del otro atributo. En ‘cantidad resultados seleccionados’ se encuentra el valor cero, que corresponde a sesiones de búsqueda donde se selecciona ningún resultado. Para estas sesiones de búsqueda, en particular, predominan consultas clasificadas como no frecuentes como lo muestra la Figura 17.

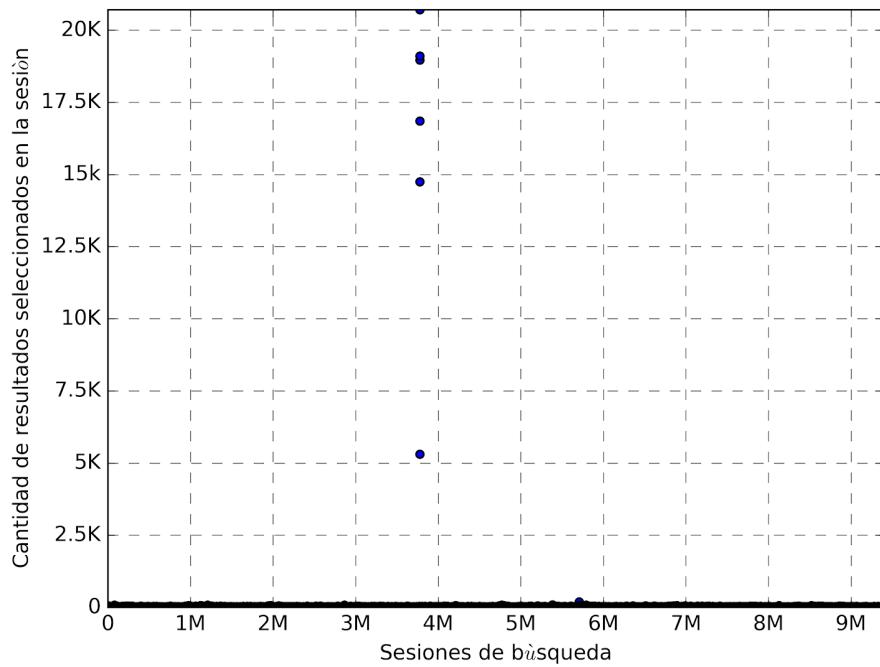


Figura 16: Cantidad de resultados seleccionados por sesión de búsqueda

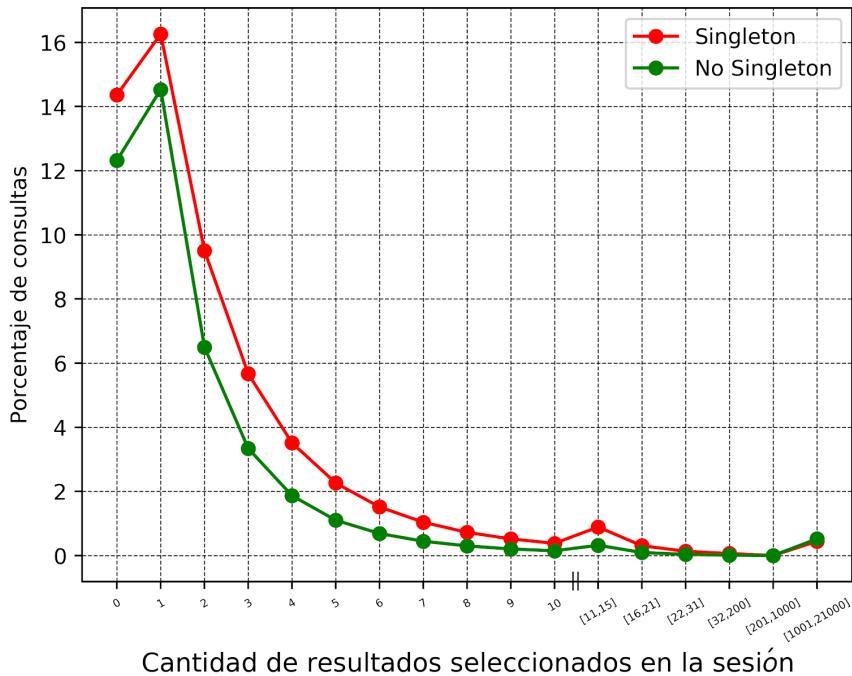


Figura 17: Relación entre la clase objetivo y la cantidad de resultados seleccionados.

4.1.3.1.3. Cantidad de resultados seleccionados en la primera posición del ranking

La cantidad de resultados seleccionados en la primera posición es un subconjunto de los resultados seleccionados donde se contabiliza solo aquellos que se encuentran en la primera posición del ranking. El resultado de esta contabilización por cada sesión de búsqueda se muestra en la Figura 18. La relación de este atributo con la clase objetivo se expone en la Figura 19, donde al igual que para el atributo ‘cantidad de resultados seleccionados’ se mantiene preponderancia de consultas infrecuentes para la mayoría de los rangos más bajos.

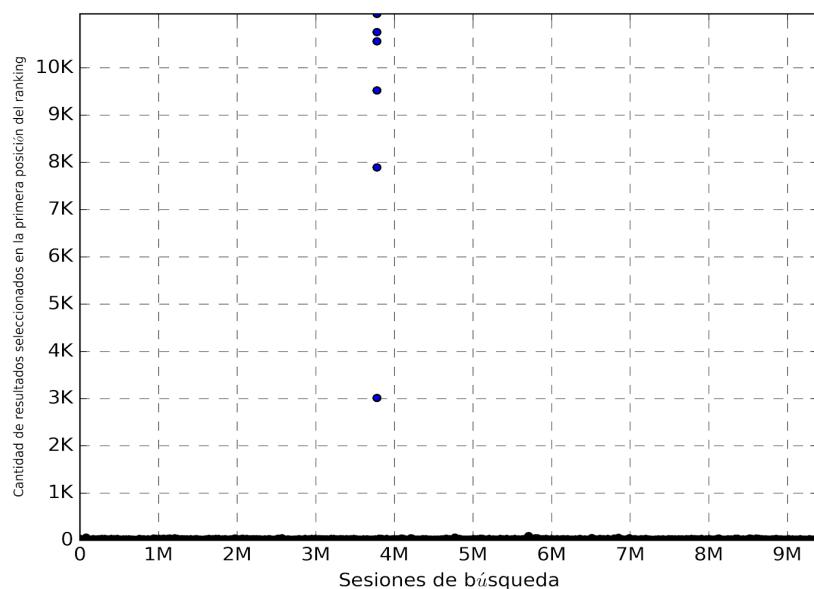


Figura 18: Cantidad de resultados seleccionados por sesión de búsqueda

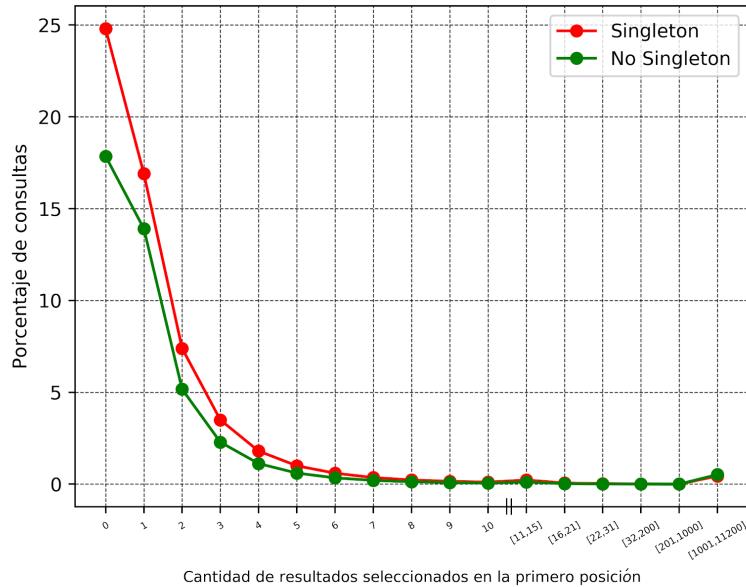


Figura 19: Relación entre la clase objetivo y la cantidad cantidad de resultados en la primera posición del ranking seleccionados.

4.1.3.1.4. Atributos de frecuencia

Los atributos presentados en esta sección describen la frecuencia de aparición de consultas y sus términos dentro en tres lapsos de tiempo diferentes.

4.1.3.1.5. Tasas de consultas

En esta sección, se estudia la frecuencia de las consultas contenidas en el query log de AOL en diferentes espacios de tiempo con el objetivo de determinar si los atributos relacionados aportan información útil sobre la popularidad de las consultas.

En la Figura 20 se muestra la tasa de consultas que arribaron al motor de búsqueda por minuto, la misma posee dos valores extremos, cercanos a 800 y 1000. El resto de los valores no supera el umbral de las 500 consultas por minuto. Por otra lado, la Figura 21 expone la frecuencia promedio de las consultas, donde se encuentran cuatro valores extremos inferiores a un valor promedio de dos. La reducida tasa de consultas produce que el lapso de un minuto sea insuficiente para percibir un posible comportamiento ráfaga o una popularidad sostenida a lo largo del tiempo. Es decir, la baja tasa de consultas por minuto dificulta realizar inferencias sólidas sobre futuras apariciones de términos y/o consultas, por lo que los atributos relacionados no tienen gran influencia en los modelos de clasificación. De acuerdo con esto, la Figura 23 y Figura 28 presentan modelos generados por C4.5 y HAT, respectivamente, en los que los

atributos relacionados con lapsos de un minuto no aparecen después del tercer nivel de los árboles y por conectados a ramas con atributos asociados a lapsos de una hora y un dia.

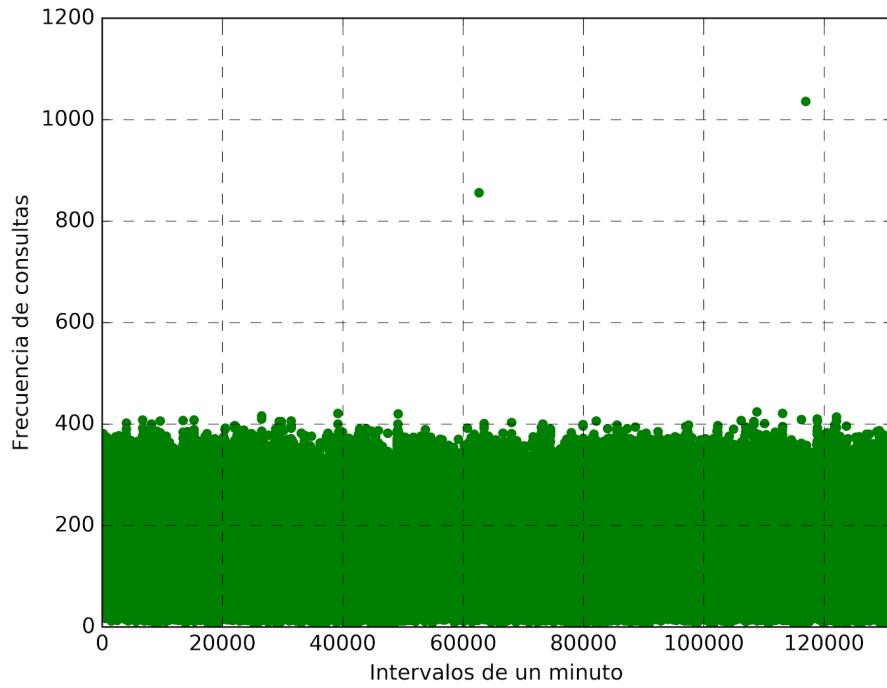


Figura 20: Tasa de consultas por minuto

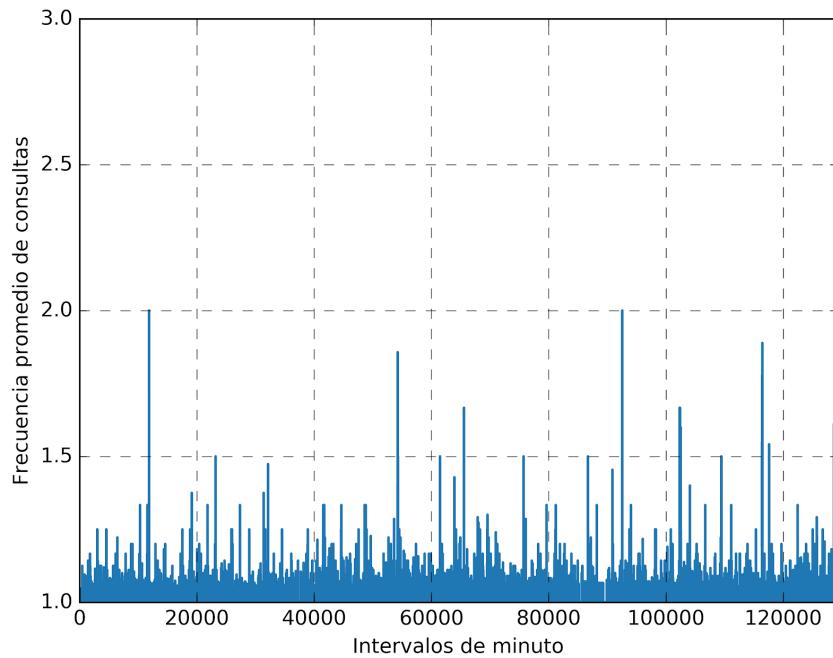


Figura 21: Frecuencia promedio de consultas por minuto

4.2. Descripción de la solución propuesta

El uso de aprendizaje máquina para administrar las políticas de admisión y desalojo a un sistema de caché ha sido propuesto anteriormente por otros autores. Sin embargo, como se comenta en la Sección 3.2.2., Tolosa [71] sugiere la exploración de intervalos óptimos para entrenar los modelos. Por esta razón, nuestra solución consiste en utilizar un algoritmo de aprendizaje continuo con mecanismos para mantenerse actualizado, evitando la necesidad de reconstruir un modelo cada cierto periodo de tiempo. En algunos casos, actualizar el modelo resulta inapropiado dado que los cambios de concepto en el flujo de consultas se pueden producir de manera abrupta, gradualmente o por el contrario, haber largos periodos de estacionalidad. Por este motivo, proponemos la utilización del algoritmo Hoeffding Adaptive Tree, que mediante el uso de ADWIN es capaz de encontrar el tamaño de ventana óptimo para adaptarse al concepto imperante en el stream de consultas. De esta forma, se utiliza al árbol decisión adaptivo para mantener las políticas de admisión a caché, en combinación con algunas de las políticas de reemplazo clásicas.

En el nivel de jerarquía asociado a caché de resultados los atributos extraíbles son más abundantes [72]. Por esta razón, este trabajo se enfoca en la mejora de una caché de resultados con la cual es posible explorar mejor el potencial de Hoeffding Adaptive Tree. Específicamente, los atributos con los que entrena a los clasificadores en este trabajo se exhiben en la Tabla 2.

Tabla 2: Lista de atributos con el que se entrena los modelos clasificación generados por los árboles de decisión.

Cantidad de caracteres	Número de términos	Presencia de URL
Presencia de errores de ortografía	Largo promedio de términos	Número de ranking
Hora de la consulta	Cantidad resultados seleccionados	Cantidad resultados seleccionados en la primera posición
Frecuencia máxima de los término de la consulta por dia	Frecuencia mínima de los términos de la consulta por dia	Frecuencia promedio de los términos de la consulta por dia.
Frecuencia máxima de los término de la consulta por hora	Frecuencia mínima de los términos de la consulta por hora	Frecuencia promedio de los términos de la consulta por hora
Frecuencia máxima de los término de la consulta por minuto	Frecuencia mínima de los términos de la consulta por hora	Frecuencia promedio de los términos de la consulta por minuto
Frecuencia de la consulta por dia	Frecuencia de la consulta por hora	Frecuencia de la consulta por minuto

La transformación de los atributos originales del query log se realiza en modo fuera de línea para simplificar la ejecución de los experimentos. Sin embargo, para que el modelo construido por Hoeffding Adaptive Tree se conserve actualizado requiere que se le suministre continuamente con nuevas consultas de ejemplo. Para ello, se propone el uso de Apache Storm⁶ por ser este un framework desarrollado específicamente para el procesamiento de flujos de datos y se describe la topología proyectada para llevar a cabo el tratamiento del stream de consultas. Como salida de la topología son obtenidas tuplas que el árbol de decisión toma para el mantenimiento del modelo de clasificación.

Una topología se ejecuta en forma distribuida en múltiples nodos trabajadores, entre los que Storm distribuye las tareas. A un nivel más bajo, los nodos contienen procesos trabajadores (*worker processes*) que se ejecutan independientemente en instancias de una Java virtual Machine (JVM). A su vez, en los worker processes corren uno o más ejecutores (*executors*) o hilos. Con este conjunto de componentes, es posible explorar la granularidad del paralelismo que para esta tarea en particular permita optimizar los recursos del cluster sobre el que se ejecuta la topología.

La Sección 2.4.2, describe los elementos básicos sobre lo que se abstrae la lógica de las tareas que realiza una topología Storm. En la solución que proponemos, un Spout toma las consultas directamente del flujo que arriba a un motor de búsqueda, construye una tupla y luego la emite dentro de la topología. Storm permite administrar estratégicamente la forma en que las tuplas son dirigidas a los Bolts en el flujo de trabajo, contando para esto con seis alternativas agrupación:

- **Shuffle:** se envían tuplas a los Bolts de forma aleatoria, en una secuencia del tipo round-robin⁷.
- **Field:** se envían tuplas a los Bolts de acuerdo al valor de uno o más atributos. Se utiliza para segmentar un flujo y conteos de tuplas de tipo específico.
- **Custom:** permite una implementación customizada de la secuencia de procesamiento de las tuplas para hacer frente a factores como carga de trabajo desigual o variaciones estacionales.

⁶ <http://storm.apache.org/>

⁷ Distribución ordenada y equitativa, donde se trata a todos los elementos con la misma prioridad.

- **Direct:** el origen decide específicamente que Bolt recibe una tupla.
- **Global:** un Bolt o Spout origen envían las tuplas generadas por todas sus instancias a una única instancia destino.

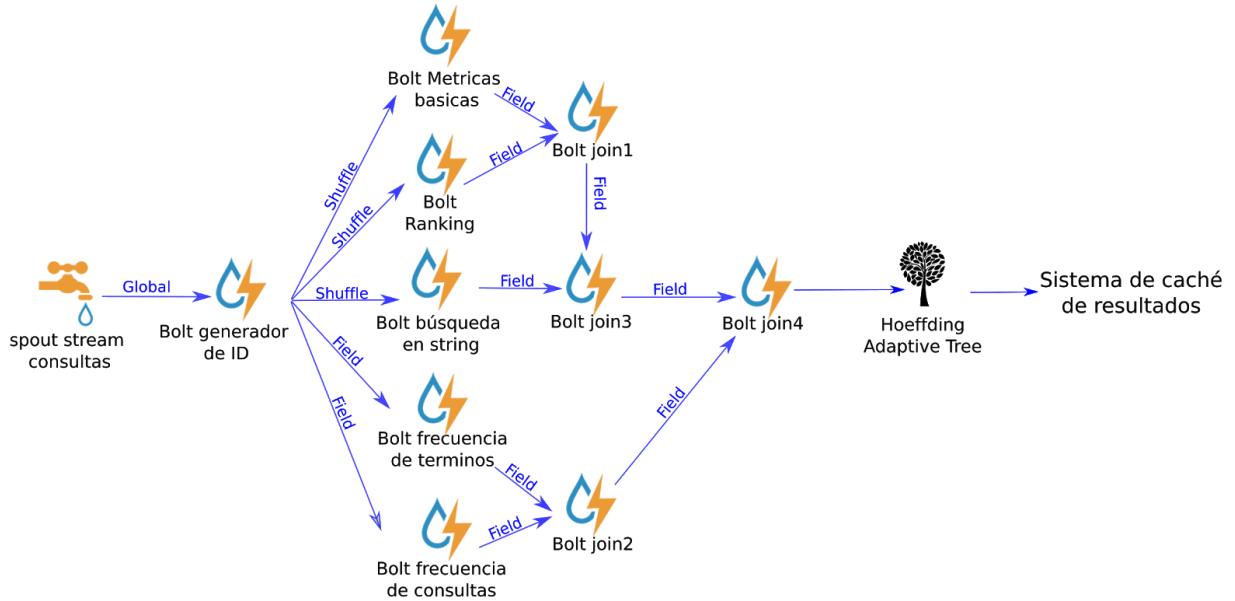


Figura 22: Topología Apache Storm para el procesamiento de un flujo de consultas. La salida de la topología es utilizada por Hoeffding Adaptive Tree para la adaptación continua del modelo al concepto vigente en el flujo.

En la Figura 22, se exhibe la topología propuesta, especificando cómo se agrupa el flujo de consultas conforme a la tareas que realiza cada Bolt. A continuación, se detallan esas tareas:

- **Bolt generator de ID:** se asigna un ID por consulta y un identificador de la sesión de búsqueda que permite unir resultados parciales en Bolts posteriores.
- **Bolt métricas básicas:** obtiene los atributos más básicos que describen a una consulta: hora de la consulta, cantidad de caracteres, número de términos y su largo promedio.
- **Bolt ranking:** obtiene atributos relacionados con los resultados seleccionados desde el ranking por el usuario: posición en el ranking, cantidad de consultas realizadas, cantidad de resultados seleccionados y cantidad de resultados seleccionados en la primera posición.
- **Bolt búsqueda en string:** se analiza la cadena de caracteres que compone a la consulta en busca de palabras con errores de ortografía y dominios de internet.
- **Bolt frecuencia de consultas:** obtiene la frecuencia de los términos en las consultas sobre el último minuto, hora y día.

- **Bolt frecuencia de términos:** obtiene la frecuencia de las consultas sobre el último minuto, hora y día.

La tarea a realizar en los últimos dos Bolt listados, está asociada con uno de los problemas clásicos en el procesamiento de flujo de datos, relacionada con encontrar los elementos de mayor frecuencia [73]. Para volúmenes de datos reducidos, la estrategia a utilizar resulta trivial, dado que una implementación simple como la que se encuentra entre los ejemplos básicos de Apache Storm resulta suficiente (Rolling Count Bolt⁸). Esa implementación, puede ser incluso adecuada para el cálculo de frecuencias de términos y consultas en el orden de minutos. Sin embargo, para el resto de los cálculos se requiere una solución que tenga límites claros en los requerimientos de memoria. Una posible alternativa, es la utilización de la estructura de datos probabilística denominada Count-min Sketch [74], que permite entre otros valores, obtener la frecuencia aproximada de los elementos dentro del stream de datos con gran velocidad. La solución implementada debe dar servicio en tiempo real a los Bolt que consultan frecuencias de términos y consultas. Por lo tanto, la base de datos Redis⁹ es una buena alternativa por dos motivos. En primer lugar, porque la base de datos cuenta con un módulo que implementa Count-min Sketch. El segundo motivo, es porque utiliza un almacenamiento de datos clave-valor en memoria, lo que permite obtener buen rendimiento y baja latencia en implementaciones no distribuidas [75]. Respecto a la última afirmación, se considera conveniente para una primera versión una implementación centralizada del sistema de mantenimiento de frecuencias.

Una vez que los Bolts asociados a la transformación y obtención de atributos procesan una tupla es necesario unirlos en una sola, para finalmente entregarla al árbol de decisión. Para ello, Apache Storm tiene predefinido un Bolt que unifica los resultados parciales mediante un atributo en común. En este caso, el Bolt generador de ID asocia cada tupla con un identificador único de consulta y de sesión de búsqueda, para obtener como salida de la topología una tupla con los atributos listados en esta sección. En el escenario planteado, podría ocurrir que al producirse un error, una tupla espere indefinidamente la llegada de otra con la que debe ser acoplada. Para evitar esta situación, es posible especificar una ventana de tiempo a esperar antes de que la tupla sea descartada.

⁸ <https://github.com/nathanmarz/storm-starter/blob/master/src/jvm/storm/starter/bolt/RollingCountBolt.java>

⁹ <https://redis.io/>

5. Experimentos y Resultados

5.1. Definición de métricas

En algunas ocasiones, la exactitud obtenida por un clasificador por sí sola, resulta una métrica exigua para evaluar la performance [76]. Por este motivo, para profundizar el análisis se utiliza una matriz de confusión (también llamada tabla de contingencia).

Dado un clasificador, y una instancia existen cuatro posibles resultados de su clasificación. En el primer caso, si una instancia es positiva (en este caso, frecuente) y es clasificada como tal, se cuenta como un verdadero positivo; si es clasificada como infrecuente, se cuenta como falso negativo. Por el contrario, si es infrecuente y es clasificada análogamente, se cuenta como verdadera negativa. Finalmente, si es infrecuente y es clasificada como frecuente, se cuenta como falso positivo. La Tabla 3, muestra una matriz de confusión para un clasificador de dos clases, donde las intersecciones entre columnas de predicción y valor real se ubican los porcentajes o cantidades contabilizadas de los tipos de resultados. A partir de estos cuatro resultados posibles, se pueden calcular diversas métricas, como por ejemplo, tasa de aciertos y precisión.

Tabla 3: Ejemplo de Matriz de confusión

		Predicción	
		Frecuente	Infrecuente
Verdad	Frecuente	Verdadero positivo (VP)	Falso negativo (FN)
	Infrecuente	Falso positivo (FP)	Verdadero negativo (VN)

Normalmente, se evalúa la performance de un clasificador mediante la exactitud o bien, usando la tasa de error. La exactitud es el porcentaje de aciertos sobre la cantidad de instancias clasificadas. Calculada en base a la matriz de confusión como:

$$\text{exactitud} = \frac{vp + vn}{vp + vn + fp + fn}$$

Por otra parte, la tasa de error corresponde a la cantidad de errores cometidos sobre la cantidad de instancias clasificadas. Calculada como:

$$\text{tasa de error} = \frac{fp + fn}{vp + vn + fp + fn}$$

Otras dos métricas que permiten ahondar en el resultado de una clasificación son la sensibilidad y la especificidad. La primera es la proporción de instancias positivas correctamente clasificados como tal. Puntualmente, para esta tarea la sensibilidad indica el porcentaje de consultas frecuentes correctamente clasificadas. Se define a la sensibilidad como:

$$\text{sensibilidad} = \frac{vp}{vp + fn}$$

Por otra parte, la especificidad es la proporción de instancias negativas que son correctamente clasificados como tal. En este caso, la métrica es un indicador de la capacidad del clasificador para predecir consultas no frecuentes. Se define a la especificidad como:

$$\text{especificidad} = \frac{vn}{vn + fp}$$

En la literatura relacionada con clasificación y la detección de señales, los errores se describen como de tipo 1 y tipo 2 [77]. El primero de ellos, íntimamente relacionado con la sensibilidad, es la tasa de falsos positivos (*false positive rate*), definida como:

$$TFP = \frac{fp}{fp + tn}$$

El error de tipo 2, se relaciona con la especificidad y se denomina tasa de falsos negativos (*false negative rate*). Se define como:

$$TFN = \frac{fn}{fn + vn}$$

5.2. Valor de referencia

La universidad de Waikato, Nueva Zelanda desarrolla en java una suite de aprendizaje automático llamada Weka. Esta suite posee una implementación del algoritmo de C4.5 de Quinlan llamada J48 [78].

Normalmente, cuando se construye un clasificador es deseable contar con un indicador de la exactitud que obtendrá el mismo frente a nuevos datos. La forma tradicional de obtener ese indicador es usando antiguos datos de los que conoce de antemano el valor que toma la clase objetivo. El conjunto de datos es fraccionado para llevar a cabo diferentes tareas, ya sea entrenamiento, validación o evaluación. Es necesario señalar que, las instancias usadas durante la etapa de entrenamiento, al ser usadas para la evaluación no serán un buen indicador de la performance futura, ya que derivaran en valores extremadamente optimistas.

Habitualmente, los algoritmos que funcionan en modo por lotes tienen una fase de entrenamiento que no puede reanudarse una vez culminada. Por lo tanto, si se requiere que el modelo incorpore nuevos ejemplos es necesario volver a construirlo desde cero. En consecuencia, la mejor alternativa para evaluar a J48 es utilizar el método clásico conocido como Holdout[79] . Este método consiste en la reserva de una cierta porción de instancias para evaluación y el uso de la porción restante para el entrenamiento del clasificador[80] . Habitualmente, se divide al dataset en una proporción 80/20, 80% de las instancias son usadas para entrenamiento y el restante 20% para evaluación. En este caso, se adopta un enfoque particular que intenta simular el funcionamiento de C4.5 frente a un stream de consultas, en el entorno de un motor de búsqueda. Por consiguiente, se entrena al modelo sobre 100 mil (100K) instancias y se usa otras 20 millones (20M) para su evaluación.

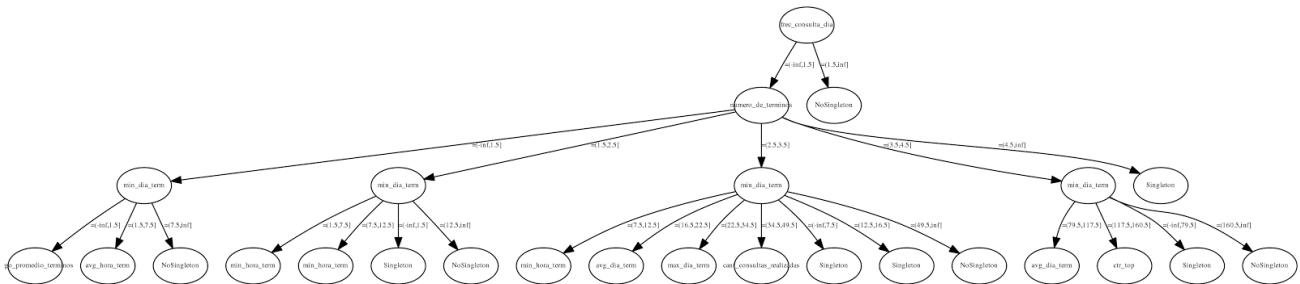


Figura 23: Recorte a 4 niveles del árbol de decisión construido por el algoritmo C4.5 a partir de 100K instancias

5.3. Desarrollo del modelo con Hoeffding Adaptive Tree

En los experimentos con el algoritmo Hoeffding Adaptive Tree se utiliza la implementación del mismo en el framework para minería de streaming de datos Massive Online Analysis [81] (MOA), al igual que Weka, desarrollado por la universidad de Waikato.

Para evaluar el algoritmo se utiliza una técnica predictiva secuencial [82] (*sequential evaluation*). Este método consiste en la utilización de cada instancia primero para evaluación y luego para el entrenamiento del modelo. De esta forma, el modelo es siempre evaluado con instancias que no ha visto previamente y es posible graficar la evolución del proceso de aprendizaje. Particularmente, la curva de evolución del proceso de aprendizaje resulta una herramienta para detectar cambios de concepto, sobreajuste del modelo, etc. En el uso de HAT frente a un stream de consultas, la razón que justifica el uso de la evaluación predictiva secuencial es la posible existencia de desvíos de concepto en la distribución de los datos, que resultan indetectables con un método de evaluación del tipo holdout. De otra forma, usando el último método mencionado sería necesario incrementar gradualmente el número de instancias usadas para el entrenamiento, teniendo que reconstruir desde cero el modelo por cada incremento.

5.3.1. Puesta a punto

En el aprendizaje de máquina, la optimización de hiperparámetros consiste en buscar el valor de los parámetros que frente a un conjunto independiente de datos genere un modelo que minimice una función de error predefinida [83]. Los hiperparametros permiten evaluar la sensibilidad de los parámetros y como su modificación repercute positiva o negativamente en la función objetivo.

Hoeffding Adaptive Tree es capaz de encontrar el tamaño de ventana óptimo para cada nodo del árbol, ajustándose a la tasa de cambio de los datos. El único parámetro que requiere que sea indicado es el límite de confianza admitido para la dividir una rama, con valores cercanos a cero las divisiones requieren de más instancias. No obstante, en el framework Moa, Hoeffding Adaptive Tree se extiende de Hoeffding Tree, lo que permite configurar algunos parámetros como el criterio de decisión, el método utilizado para manejo atributos numéricos, habilitar la pre poda del árbol, etcétera.

5.3.1.1. Evaluación de criterios de división

Como se explica en la Sección 3.2.3, los criterios de división son utilizados por los algoritmos de árboles de decisión para escoger qué atributo es colocado en los diferentes niveles de arbol. En este apartado se presentan los resultados de la ejecución del HAT variando los criterios de división disponibles.

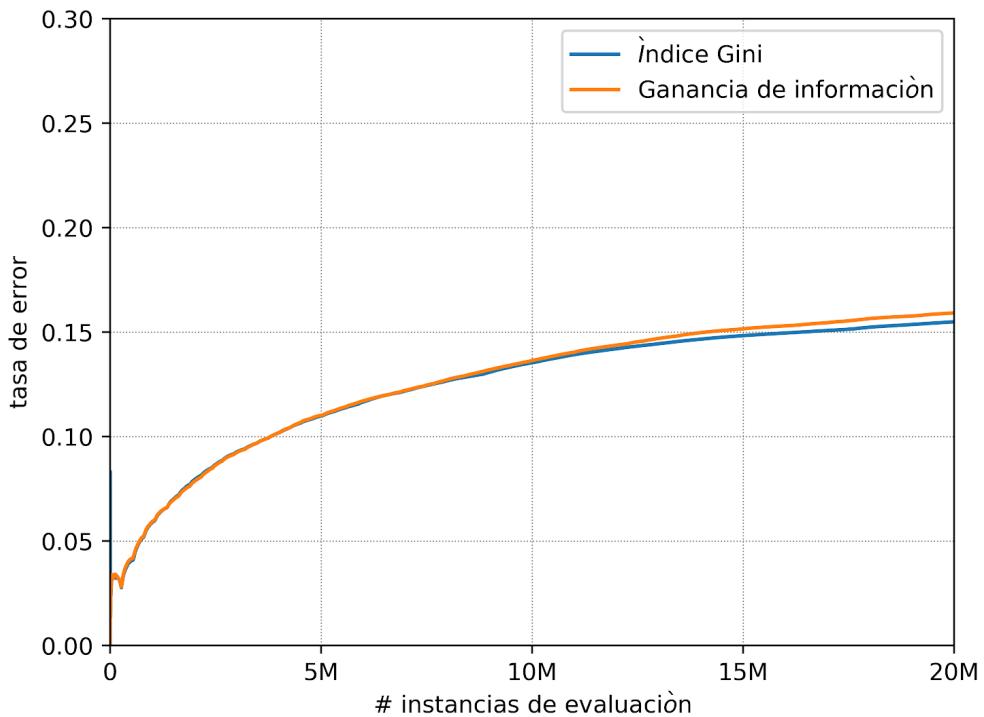


Figura 24: Comparación de tasa de error de HAT variando criterios de división, índice de Gini y ganancia de información.

El rendimiento obtenido a partir del uso del índice de Gini y la ganancia de información resulta similar. Específicamente, la diferencia entre su tasa de error no supera el 1%, como se puede apreciar en la Figura 24. Aunque el error resultante del uso de Gini es apenas inferior, la tendencia sugiere que podría acrecentarse la diferencia frente a un mayor número de instancias. Por lo tanto, se elige al índice de Gini para su uso con HAT en los siguientes experimentos.

5.3.1.2. Evaluación de métodos para manejo de atributos numéricos

Como se describe en la Sección 3.2.4.1, existen diferentes métodos para el manejo de atributos numéricos. El estimador ADWIN, componente principal en Hoeffding Adaptive Tree, opera con

atributos numéricos usando por defecto el método histograma exponencial. Sin embargo, la forma en que está implementado este algoritmo en MOA permite la utilización de otros métodos. Por consiguiente, se explora el desempeño de los métodos disponibles frente a la clasificación de consultas. La evaluación de los métodos se realiza con 20M de instancias y su rendimiento se expone en la Figura 25. Los métodos que mejor exactitud obtienen son histograma exponencial y aproximación gaussiana, con valores cercanos al 85% al finalizar la clasificación de todo el conjunto de instancias. VMFL alcanza valores de exactitud similares, con una diferencia que es menor a un 3% de exactitud. Por debajo de VMFL, con un diferencia ahora más significativa (mayor a un 5%), se encuentra el método Greenwald Khanna. La performance más baja se obtiene con Binary Tree, con solo un 60% de instancias clasificadas correctamente, casi un 25% por debajo de los métodos más exactos.

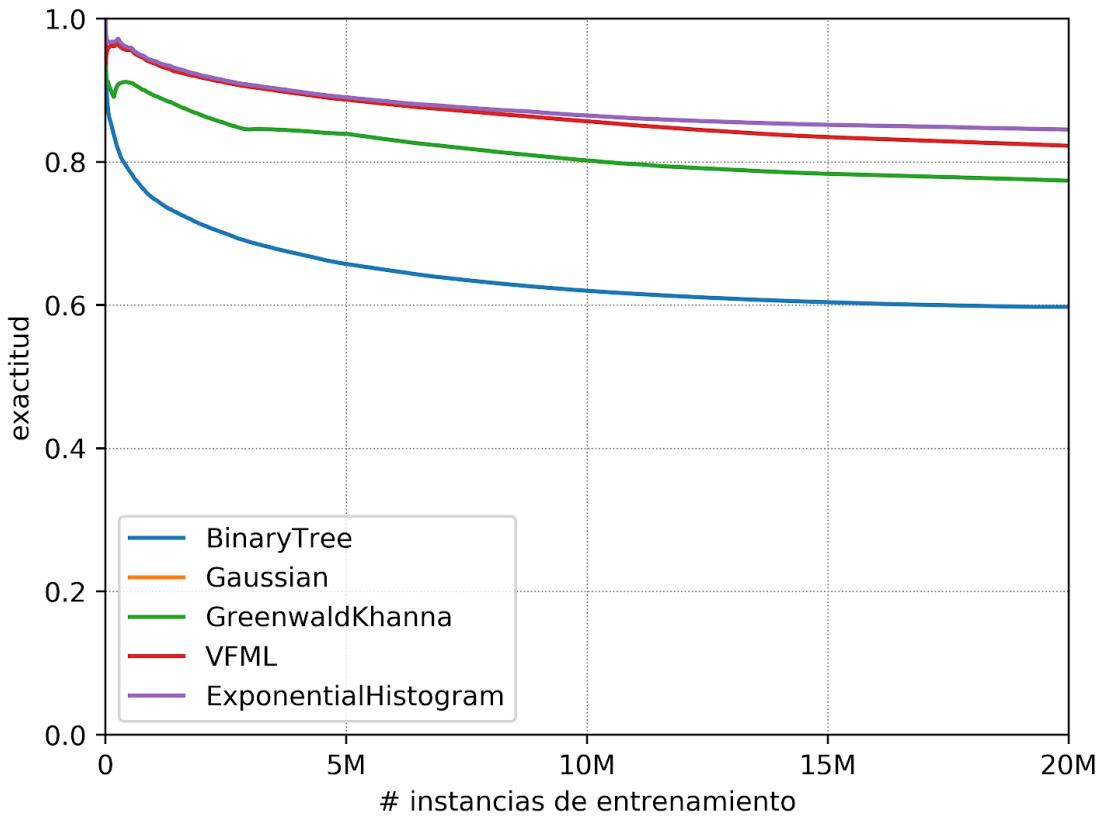


Figura 25 : Comparación de performance de clasificación de HAT variando métodos de manejo de atributos numéricos.

5.4. Comparación de rendimiento de C4.5 y Hoeffding Adaptive Tree

Para la comparación de C4.5 y Hoeffding Adaptive Tree se utiliza nuevamente el query log AOL ordenado temporalmente. El primero de ellos es entrenado en base a 100K instancias y

evaluado con un conjunto, disjunto al anterior, de 20M de instancias. Para una comparación equitativa se contrasta la performance de HAT a partir del mismo conjunto de evaluación usado con C4.5. Aunque con HAT, a diferencia del anterior, se utiliza el método de evaluación predictiva secuencial con un pequeño precalentamiento de 200 instancias. La exactitud obtenida por los clasificadores es representada en la Figura 26.

Como primera observación interesante, ambos clasificadores mantienen su exactitud por encima de una 90% al clasificar las primeras 500K instancias, aunque HAT lo mantiene hasta las primeras 3,5M primeras. Aunque ambos modelos tienen una tendencia a deteriorar su performance, C4.5 se ve más afectado conforme más instancias son clasificadas, deteriorándose hasta un 20% por debajo del valor con el que inicia y finalmente, obteniendo una exactitud del 70% al cabo de clasificar 20M de instancias. Es decir, un 30% de las instancias son incorrectamente clasificadas por C4.5 (ver Figura 27). Este porcentaje tiene significativas implicaciones en la tasa de aciertos de la caché, al ser este clasificador utilizado para administrar las políticas de admisión. Por otra parte, HAT al cabo de clasificar 20M reduce su exactitud en sólo un 10%, logrando aproximadamente un 84,5% de instancias correctamente clasificadas.

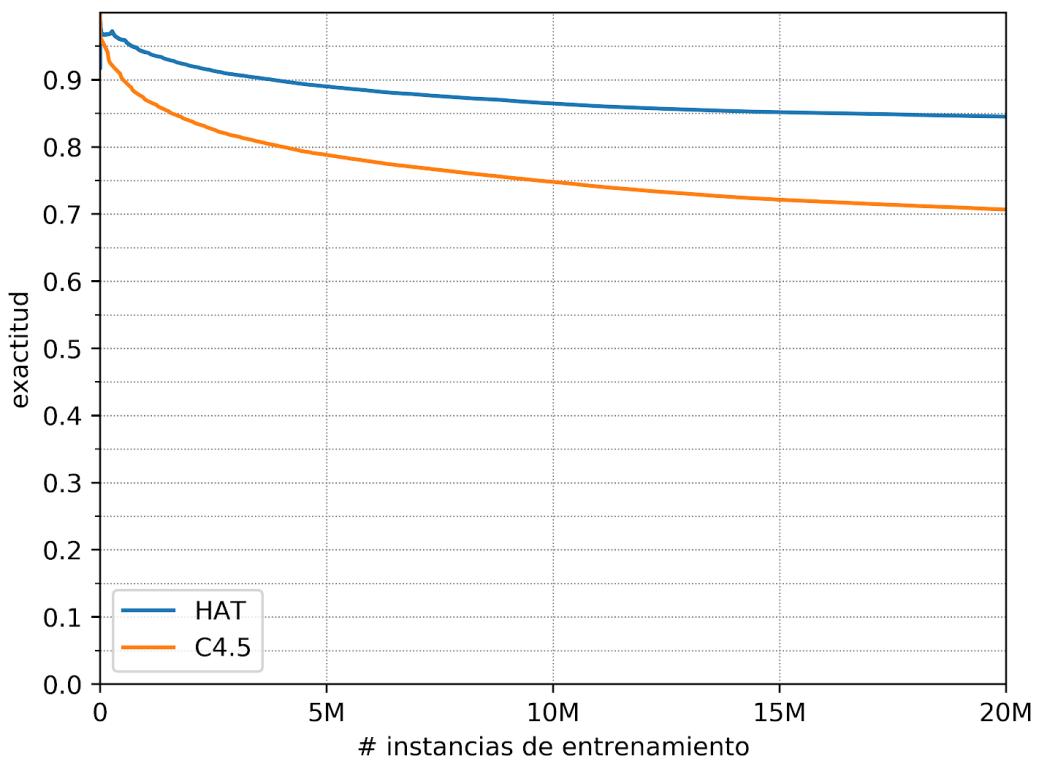


Figura 26: Comparación de la exactitud de la clasificación entre C4.5 y HAT utilizando 20M de instancias del query log de AOL.

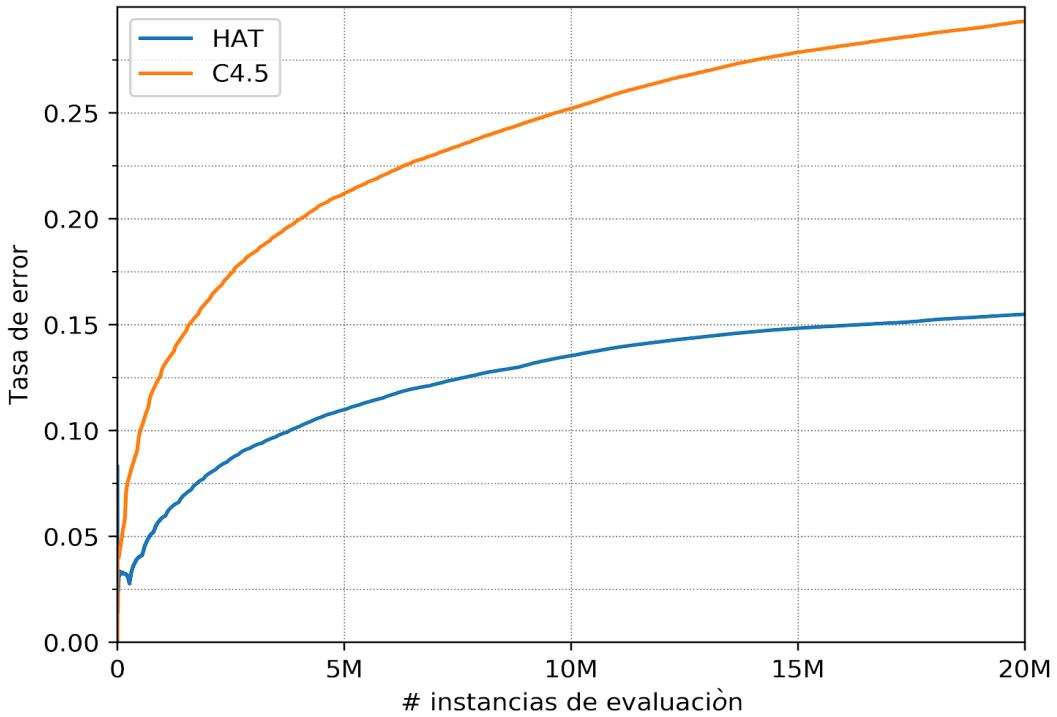


Figura 27: Comparación de la tasa de error en la clasificación entre C4.5 y HAT utilizando 20M de instancias del query log de AOL.

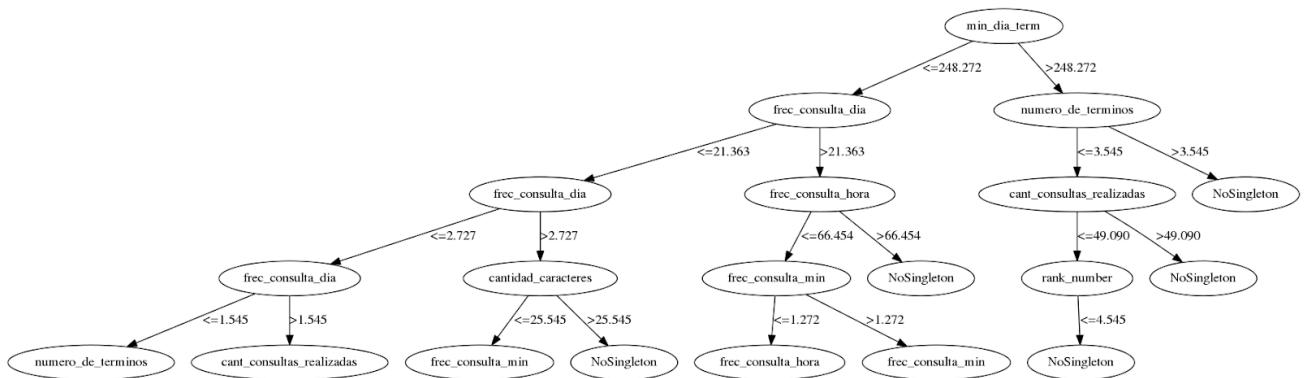


Figura 28: recorte del árbol de decisión generado por HAT a partir de 20M de instancias

La tablas de contingencia nos permite analizar individualmente un error que resulta de mayor severidad en un clasificador que es utilizado para gestionar una política de admisión. Como señala en su trabajo Tolosa [71], al clasificar un elemento como falso negativo se podría estar excluyendo indeterminadamente de la caché a un elemento que aparece recurrentemente.

Tabla 4: Matriz de confusión con los valores obtenidos en la clasificación de C4.5 a partir de la clasificación de 20M de instancias.

		Predicción	
		Frecuente	Infrecuente
Verdad	Frecuente	0.131 (Verdadero positivo)	0.292 (Falso negativo)
	Infrecuente	0,0006 (Falso positivo)	0.574 (Verdadero negativo)

Tabla 5: Matriz de confusión con los valores obtenidos en la clasificación de HAT a partir de la clasificación de 20M de instancias.

		Predicción	
		Frecuente	Infrecuente
Verdad	Frecuente	0.311 (Verdadero positivo)	0.111 (Falso negativo)
	Infrecuente	0.043 (Falso positivo)	0.533 (Verdadero negativo)

Como primera observación, basada en la Tabla 4 y la Tabla 5, ambos clasificadores tienen una mejor especificidad que sensibilidad. Es decir, ambos identifican mejor a consultas no frecuentes, aunque la especificidad de C4.5 es superior a la de HAT, que es acompañada por una tasa de falsos positivos baja (aproximadamente un 0,1%). En cambio, al clasificar consultas frecuentes HAT se desempeña mejor, alcanzando una sensibilidad considerablemente mayor (cercana a un 18%). La misma es influida por los falsos negativos, que como se comenta antes es el error más severo para la clasificación de consultas. C4.5 tiene casi un 30% de falsos negativos contra un 11% de HAT, es decir casi tres veces superior. En consecuencia, esta relación se mantiene para la tasa de falsos negativos (o falsas consultas singleton), ya que para HAT es de un 26% versus un 69% de C4.5. Para la tarea que realizan estos clasificadores, es apreciable que HAT tenga un porcentaje menor de falsos negativos con relación a C4.5.

5.4.1. Utilización de recursos

La capacidad de HAT para aprender incrementalmente le permite adecuarse a entornos con recursos limitados o escasos. Principalmente esa capacidad radica en que es posible transferir gradualmente, desde disco a memoria, las instancias usadas para el ajuste del modelo. Para

analizar la utilización de memoria se emplea la herramienta visualVM¹⁰ que posibilita visualizar la memoria requerida por los procesos en tiempo de ejecución. Con el uso de esta herramienta se puede apreciar que la cantidad de instancias transferidas es proporcional al espacio en memoria usado. Por ejemplo, en una evaluación predictiva secuencial de HAT, con lecturas de disco de 200K instancias por vez, se requiere alrededor de un GB de memoria. Ahora bien, con lecturas más pequeñas, de solo 100 instancias por vez, se registra un consumo máximo de 300MB. La Figura 29 y Figura 30 exhiben el consumo de memoria con lecturas de 100 y 200K instancias, respectivamente.

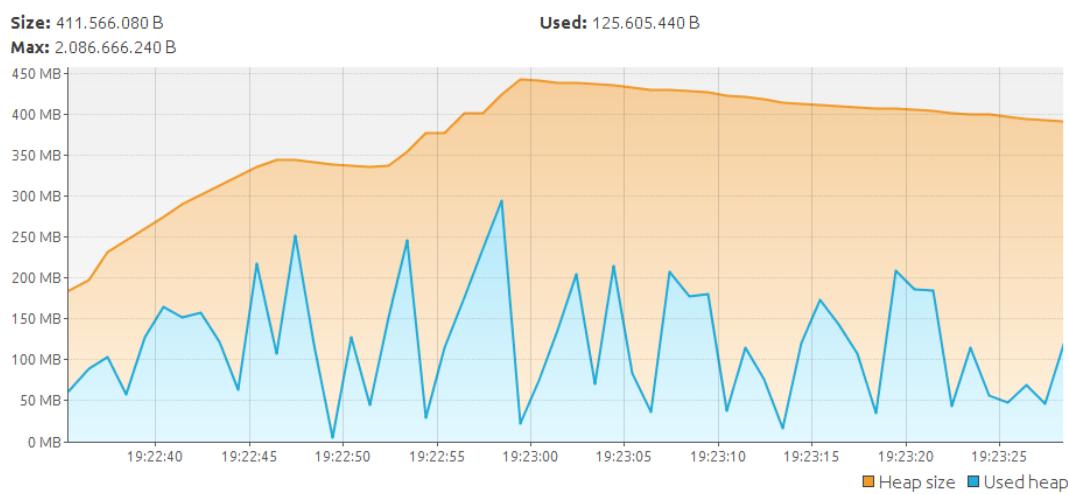


Figura 29 : Consumo de memoria en la ejecución de evaluación predictiva secuencial con lecturas de 100 instancias

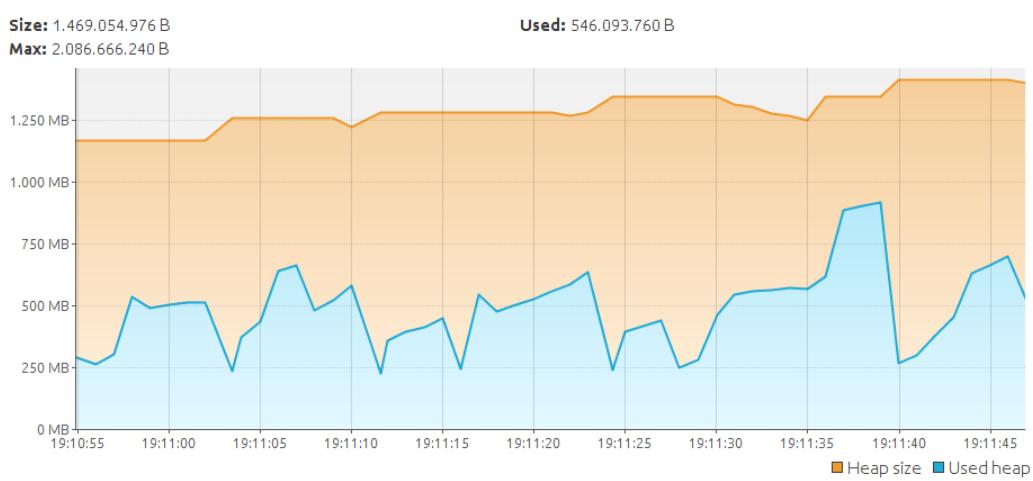


Figura 30 : Consumo de memoria en la ejecución de evaluación predictiva secuencial con lecturas de 200K instancias

¹⁰ <https://visualvm.github.io/>

Usando un algoritmo de aprendizaje que tiene un funcionamiento por lotes la cantidad de memoria utilizada no es alterable mediante la cantidad de instancias que son leídas de un archivo, una base de datos, la API de una red social o cualquier otra fuente de datos. J48 es una algoritmo de aprendizaje en modo por lote y, por lo tanto, al momento del entrenamiento todas las instancias deben ser transferidas a memoria para que el algoritmo realice la construcción del modelo. De esta forma, con este tipo de algoritmos si el conjunto de datos de entrenamiento excede la capacidad de memoria disponible se requiere la utilización de un método de muestreo [84].

5.4.2. Rendimiento frente a cambios de concepto

Ambos algoritmos de árboles de decisión utilizados, al igual que en la tesis de Tolosa [71], alcanzan una performance aceptable para los requerimientos de este trabajo. Específicamente en estos experimentos se obtienen valores de exactitud superiores al 85% para 20M de instancias clasificadas. Conforme a esto, en un entorno de producción, los modelos generados nos permitirán clasificar erróneamente solo un 20% de las consultas que arriban a un motor de búsqueda.

Se debe subrayar que HAT requiere un esfuerzo para establecer su configuración considerablemente superior al requerido por C4.5. Esto se debe a que el primero de ellos incorpora la necesidad de desarrollar y mantener un sistema accesorio que le suministre los atributos de las consultas para su aprendizaje continuo y actualización del modelo al concepto vigente en el flujo de datos. HAT debido a su capacidad de adaptarse a cambios de concepto, debería tener una ventaja significativa frente a un algoritmo con funcionamiento en modo por lotes entrenado con una muestra aleatoria, en el ambiente de los motores de búsqueda. Dado que, en este ambiente se producen cambios de concepto ligados a la variabilidad de las necesidades de información de los usuarios. Los experimentos hasta ahora expuestos, muestran una superioridad por parte de HAT que justifica el esfuerzo extra a realizar. Sin embargo, en el conjunto de datos de entrenamiento es probable que no se presenten cambios lo suficientemente importantes como para que HAT evidencie su verdadero potencial. Un problema con la mayoría de los conjuntos de datos del mundo real disponibles para la experimentación es que poseen cambios de concepto diminutos [85]. Una opción para juzgar el rendimiento de HAT en un entorno realmente adverso es introducir artificialmente cambios de concepto en la distribución de los datos. Este método ha sido utilizado por otros autores para cuantificar la capacidad de los algoritmos para adaptar los modelos conforme se suceden los cambios [86]. Puntualmente, para

evaluar a HAT se modifica la forma en que es definida la clase, dejándola solo ligada a los valores que tome un atributo. Dicho de otra forma, se introduce una dependencia completa hacia el valor de un atributo por parte de la variable objetivo. En este caso, se escoge el atributo ‘cantidad de términos’. El experimento consiste en tomar una cierta cantidad de instancias (20M), y clasificar como frecuentes (no singletons) sólo a aquellas con cantidad de términos mayor a tres. Este atributo es elegido por dos motivos. El primero, la cantidad de términos es ponderada para la asignación de la clase por C4.5 y HAT. Dado que el atributo se ubica en el segundo y tercer nivel del árbol respectivamente, como se puede ver en la representación en la Figura 23 y la Figura 28 . Al introducirse un cambio de concepto, la ubicación cercana a la base del árbol obliga a que el modelo desactualizado deba cambiar radicalmente para poder capturar la nueva tendencia. El segundo motivo es que el cambio planteado podría producirse en un ambiente real. Por ejemplo, podría originarse una inclinación de los usuarios sobre un tema que requiere ser consultado con un número de términos superior a tres.

Para generar el conjunto de datos de evaluación utilizado en este experimento, se repiten exactamente 20M de instancias del dataset y sobre la copia se modifica el valor de la clase objetivo en base al del atributo cantidad de términos.

Una vez que se introduce el cambio de concepto, la adaptación de HAT se percibe antes de las 100K instancias, apreciable en la Figura 32. Como se esperaba, el algoritmo actualiza el modelo para ajustarse a la nueva condición que define la clase. Para hacerlo éste poda antiguas ramas, eliminando reglas inválidas, hasta quedarse con solo la raíz que contiene la regla determinada por el atributo ‘cantidad de términos’. Al modificarse correctamente las reglas que definen la clase, la exactitud de clasificación comienza un crecimiento continuo. La conformación final del modelo es representada en la Figura 31. A diferencia de HAT, C4.5 no tiene la capacidad de adaptarse a los cambios y su modelo aún se encuentra ajustado al antiguo concepto. Por lo tanto, el clasificador C4.5 comienza a deteriorar su rendimiento en cuanto enfrenta a las instancias con el desvío.

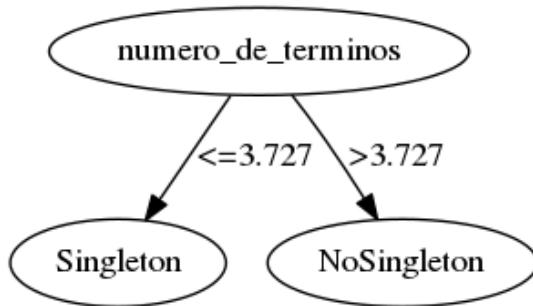


Figura 31 : Modelo resultante de la adaptación de HAT al cambio de concepto en la distribución de las consultas.

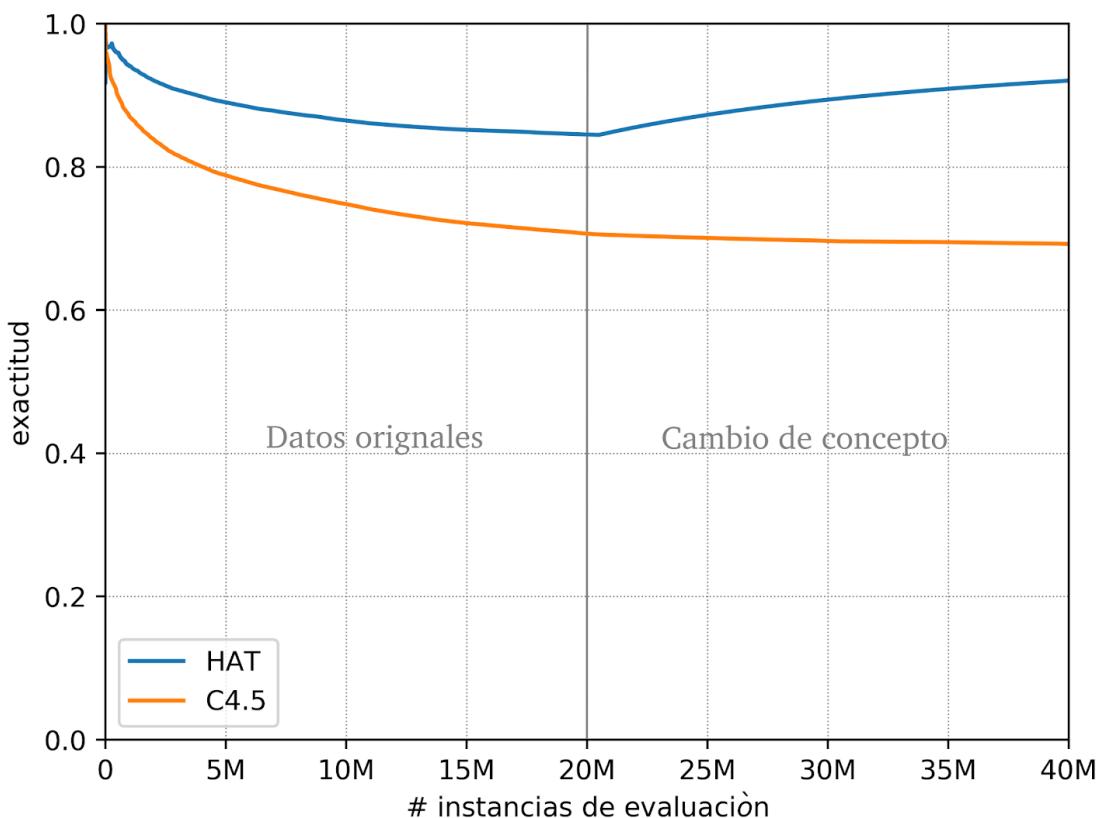


Figura 32 : Evaluación de HAT y C4.5 frente a un cambio de concepto. A partir de 20M se asigna la clase No Singleton solo a las consultas con más de cuatro términos.

5.4.3. Evaluación del modelo en combinación con políticas de desalojo dinámicas

Como se describe en la Sección 3.2, las políticas de caché se agrupan en dos clases: estáticas y dinámicas. En la primera clase, la caché tiene un tamaño fijo y es llenada con elementos previamente computados, los cuales no modifican su estado durante la ejecución. A diferencia

de éstas, en las políticas dinámicas la memoria se llena en línea y los elementos en ella pueden ser reemplazados bajo algún criterio. En este caso, se utiliza un algoritmo de árboles decisión que persigue adaptarse a los cambios que se producen normalmente en la distribución de las consultas. El algoritmo HAT genera, a partir de instancias vistas, un modelo dinámico para decidir qué consultas deben ser admitidas en la caché. Sin embargo, el tamaño de la caché no es infinito y cuando esta esté llena, se debe decidir qué elemento desalojar para dar lugar a nuevos. Es decir, en esta configuración HAT se encarga de generar las reglas para la admisión de consultas. Por otro lado, para el desalojo se experimenta con algunas de las estrategias clásicas. En esta sección, se exponen los resultados obtenidos con algunas de las estrategias de reemplazo más convencionales.

Existen diferentes acercamientos para decidir qué elementos deben ser desalojados de la memoria caché, los autores Podlipnig y Böszörmenyi[87], realizan un relevamiento de diferentes técnicas propuestas en la literatura y resumen los factores más importantes usados en el proceso de reemplazo:

- Frescura: tiempo desde la última vez que un objeto fue utilizado.
- Frecuencia: número de peticiones de un objeto.
- Tamaño: tamaño de un objeto.
- Costo: costo asociado a la obtención de un objeto.
- Tiempo de expiración: tiempo que tarda un objeto en volverse viejo, en el que debe ser reemplazado inmediatamente.

En los experimentos de evaluación del rendimiento de caché se utilizan cuatro políticas de reemplazo asociados a factores de frecuencia y frescura:

First In First Out (FIFO): Este es una de las estrategias más ingenuas y sencillas de implementar. Se mantiene una estructura de cola y cuando la caché está llena se desaloja a aquel elemento que primero ingresó, sin tener en cuenta que tan frecuentemente es utilizado o cuantas veces se lo solicitó.

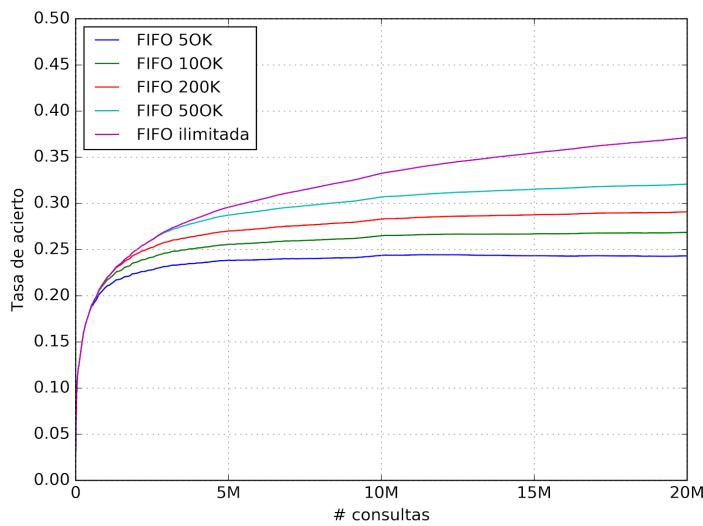
Last In First Out (LIFO): esta estrategia tiene el comportamiento exactamente opuesto al de FIFO, se desaloja al último elemento que ingresó, sin tener en cuenta que tan frecuentemente es utilizado o cuantas veces se lo hizo.

Least Recently Used (LRU): se escoge desalojar a aquel ítem con mayor tiempo desde su última solicitud.

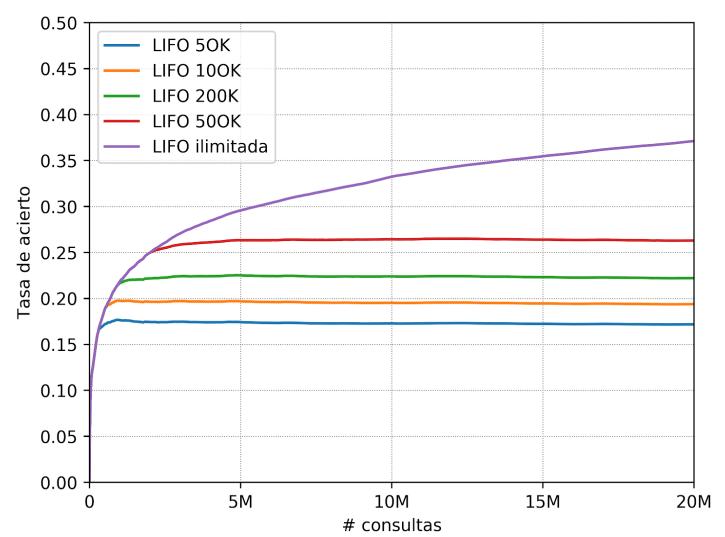
Least Frequently Used (LFU): Se mantiene un contador con la frecuencia de veces que el ítem es requerido. En caso de requerirse espacio para un nuevo elemento, se remueve el ítem con la frecuencia de solicitud baja.

Para llevar a cabo los experimentos se entrena incrementalmente al modelo generado por HAT, manteniendo el acercamiento antes usado en las evaluaciones predictivas secuenciales. De acuerdo con esto, si una consulta es clasificada como frecuente se la almacena en caché y posteriormente la misma es usada para entrenar al modelo. En este caso, el nivel de jerarquía de caché se corresponde con el de caché de resultados. En una implementación tradicional de este tipo se almacenen los top-k resultados asociados a una consulta. Por esta razón, en este experimento se asume que cada elemento almacenado en caché requiere un espacio constante. Bajo esta premisa, se calcula la tasa de aciertos variando las capacidades de caché y simulando el almacenamiento de los resultados asociados a N consultas diferentes. Inicialmente, las capacidades con las que se evalúa son 50K, 100K, 200K, 500K y una memoria caché teórica de tamaño infinito.

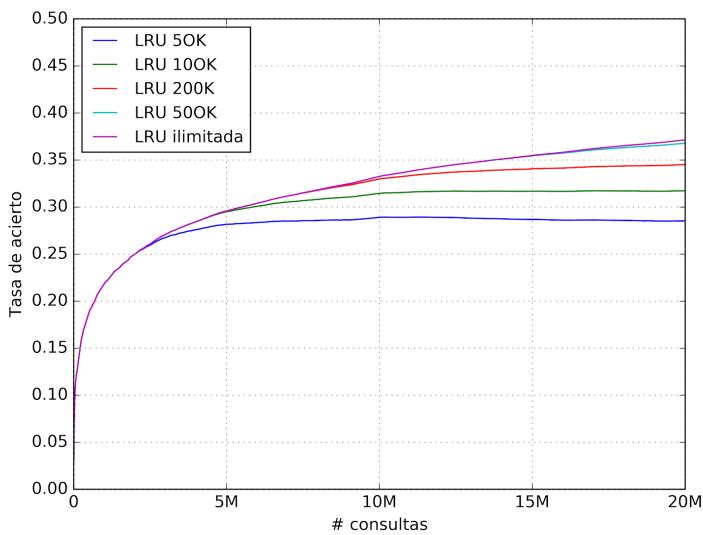
Resulta relevante conocer el porcentaje máximo de aciertos que se puede alcanzar con una política de reemplazo óptima. De esta forma, se puede cuantificar cuánto es posible mejorar el rendimiento de una política o también, cuál es la ganancia que se obtiene al incrementar el tamaño de la caché. Relacionado con esto último, podría suceder que la mejora obtenible no justifique los costos de una inversión en hardware para expandir el tamaño de la memoria caché o, por el contrario, que sí este justificada la inversión. Puntualmente en estos experimentos, para encontrar el techo del rendimiento se simula la existencia una memoria de tamaño infinito, con la cual se obtiene la tasa máxima de aciertos alcanzable por la política de admisión, que bajo esta configuración es de un 37%. Cabe aclarar que ese valor es solo mejorable modificando la política de admisión. Por lo tanto, la tasa de aciertos con caché infinita resulta idéntica para todas las políticas de reemplazo utilizadas y se incluye en todos los gráficos como una referencia.



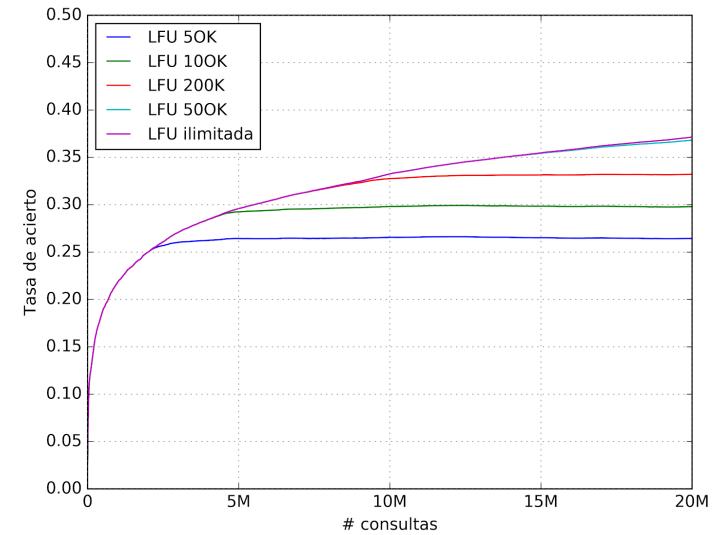
(a)



(b)



(c)



(d)

Figura 33: Evaluación de la tasa de aciertos variando la políticas de reemplazo y capacidades de caché, en combinación con HAT como política de admisión

Con todas las de capacidades de memoria, LRU es la política de reemplazo que mejor performance (ver Figura 34). La segunda política que mejor rendimiento tiene es LFU, con una

diferencia de poco menos del 2% que la anterior al cabo de la clasificación de 20M de instancias del query log. Al usar un tamaño de caché de 500K, LRU y LFU obtienen un porcentaje de aciertos similar al que se obtiene con memoria infinita, siendo inferiores solo en un 0,3%. Sin embargo, la performance de FIFO y LIFO es significativamente más baja, ya que la diferencia entre el máximo teórico y la capacidad de 500K para ambas es de más de un 5%. El rendimiento más pobre corresponde a LIFO, con la máxima capacidad de caché evaluada, no alcanza el 30% de aciertos. Más allá de esto, resulta llamativo que al llenarse la caché la cantidad de aciertos y desaciertos se mantiene con poca varianza a medida que nuevas consultas son clasificadas, este comportamiento se muestra en la Figura 33 (b).

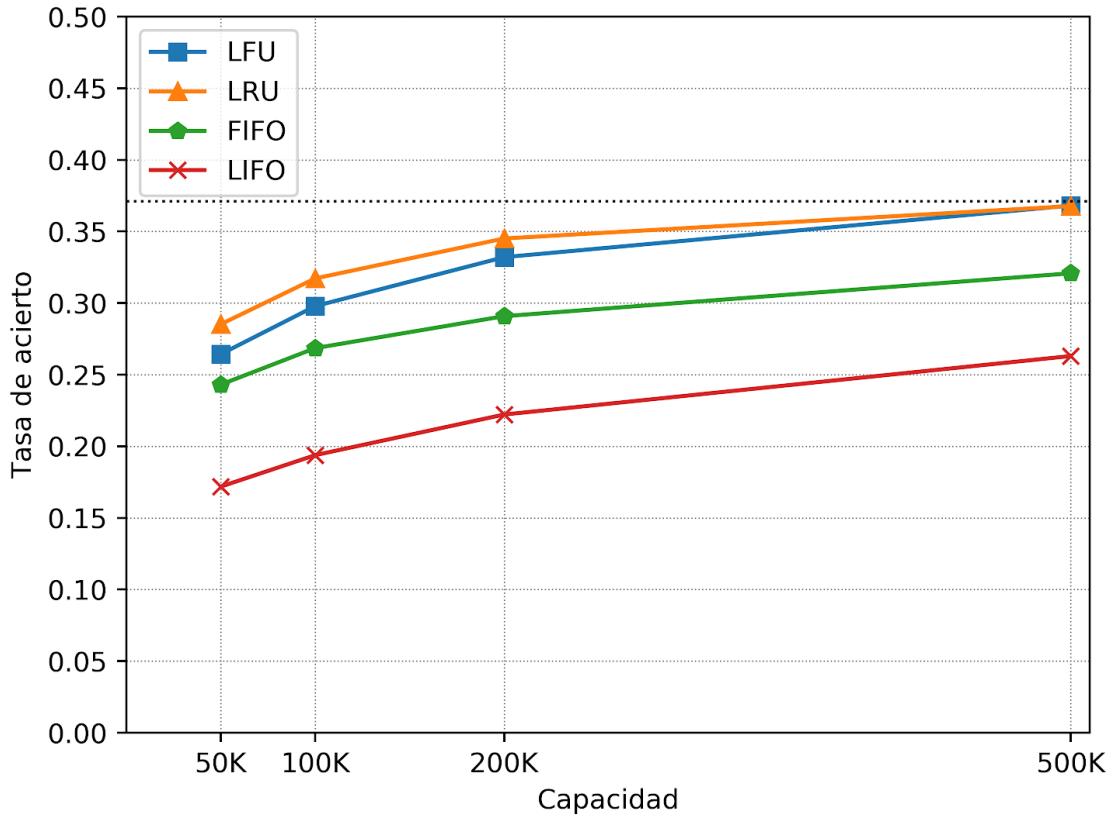


Figura 34: Comparación de la tasa de aciertos obtenidos con las diferentes políticas de reemplazo. La línea negra con más intensidad representa el umbral de tasa de acierto

Antes se muestra el rendimiento de HAT simulando la existencia de una política de reemplazo óptima. En la misma línea, se confronta la performance de los modelos generados por C4.5 y HAT, ahora con una política de admisión óptima teórica. Para esta comparación en la literatura [71,88,89] se utiliza un algoritmo clarividente o oráculo (*Bélády's Algorithm*). El mismo posee la capacidad de incorporar a la caché un elemento sólo cuando efectivamente volverá a ser

requerido y desalojar aquellos que no se presentaran nuevamente. Tomando la idea de Kucukyilmaz [89], se plantean dos variantes del algoritmo mencionado, un oráculo teórico y oráculo práctico. El oráculo teórico conoce por adelantado el flujo de consultas y por ende, solo almacena consultas que serán realizadas nuevamente en el futuro. Aunque resulta considerablemente irrealista la suposición de que un sistema posea almacenado en caché un elemento sin haberlo procesado por lo menos una vez. El oráculo práctico, a diferencia del anterior, brinda una comparación más análoga a un entorno real, simulando tener almacenada una consulta a partir de su segunda aparición.

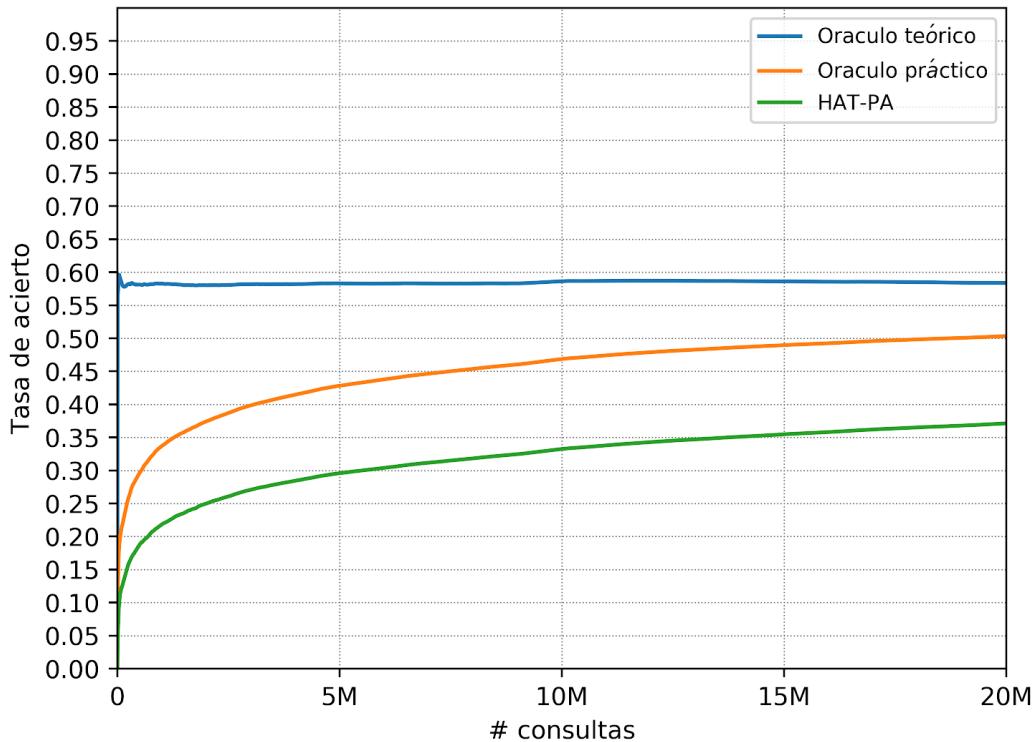


Figura 35: Comparación del política admisión administrada por HAT contra el oráculo de teorico y práctico.
HAT-PA: Política de admisión construida con HAT.

Entre la tasa de aciertos del oráculo teórico y práctico existe una diferencia de aproximadamente un 7%. Por otro lado, el modelo de admisión dinámico generado con HAT tiene una tasa de aciertos un 13% por debajo de la que obtiene el oráculo práctico.

En una comparación más detallada de C4.5 y HAT, se calcula la tasa de aciertos alcanzada por estos con capacidades de caché que varían en un rango de [1K - 1M], en combinación con la política de desalojo que mejor performance obtiene en los experimentos (LRU). Frente a esta configuración, C4.5 obtiene una tasa de acierto idéntica (de 18,93%) para el rango [10K-1M], apreciable en la Figura 36 y Figura 37 . La tasa de aciertos resulta constante para el rango debido

a que el clasificador admite solo 17K consultas diferentes (contra 905K que admite HAT). Por esta razón, solo se ve el efecto de la política de reemplazo con capacidades de caché inferiores a 10K, donde C4.5 logra superar a HAT en menos de un 3% . En contraste, con capacidades de caché superiores a 10K, HAT supera ampliamente a C4.5, con diferencias de hasta un 15%.

Para capacidades de caché mayores a 10K, HAT exhibe un crecimiento importante en la tasa de acierto. Frente al incremento de tamaño de 100K a un 1M se observan mejoras de aproximadamente un 10%. Analizando la relación costo-beneficio, resulta interesante la escasa ganancia obtenida al incrementar el tamaño de caché de 500K al doble, 36,768% versus un 37,125%, diferencia que no supera el 0,4%. Se hace evidente que la mejora no es proporcional al incremento de la capacidad de caché y que en cierto punto, ampliar el espacio no posee efecto alguno en el rendimiento. En la Figura 38, se muestra el rendimiento de C4.5 y HAT con caché de tamaño infinito. En este último gráfico, se puede apreciar que HAT logra duplicar la tasa de aciertos de C4.5, en congruencia con la tendencia que presentada con capacidades de caché finitas. Si bien ambos algoritmos obtienen un exactitud en la clasificación similar (ver Figura 26), existen diferencias significativas en la tasa de aciertos.

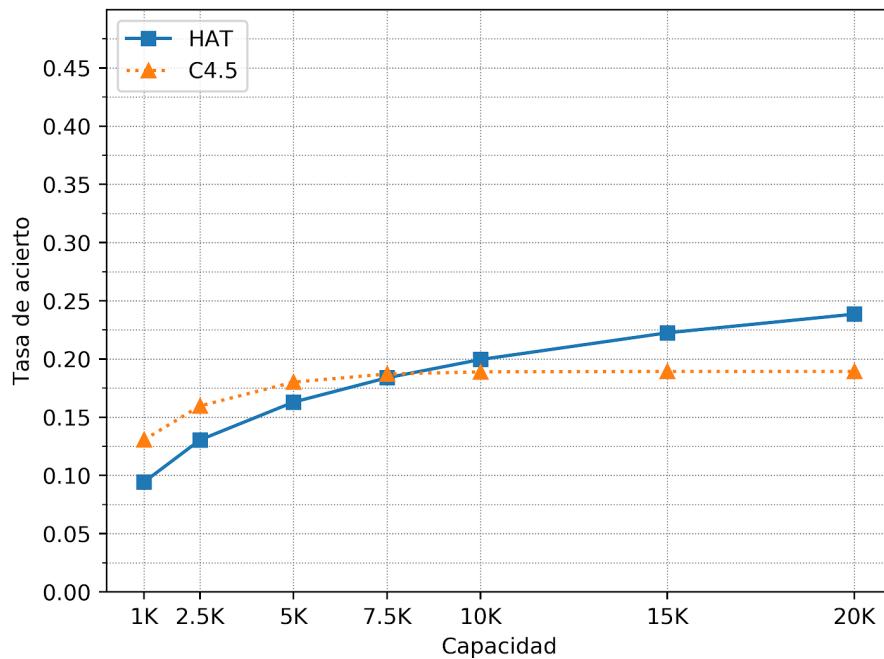


Figura 36: Comparación del rendimiento de usando C4.5 y HAT variando la capacidad de la caché en el rango 1K-20K, utilizando la política de reemplazo LRU

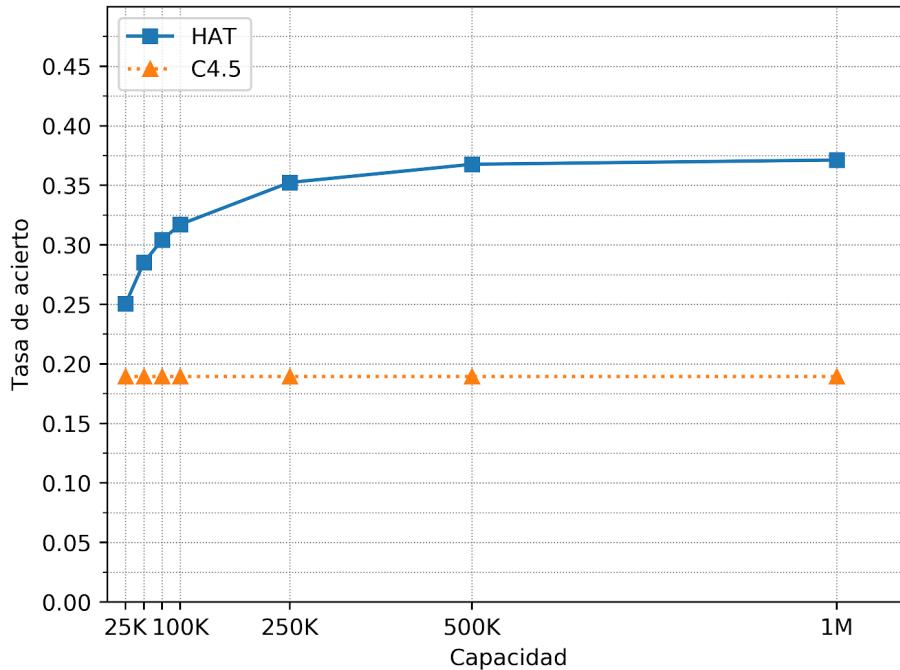


Figura 37: Comparación del rendimiento de usando C4.5 y HAT variando la capacidad de la caché en el rango 25K-1M, utilizando la política de reemplazo LRU

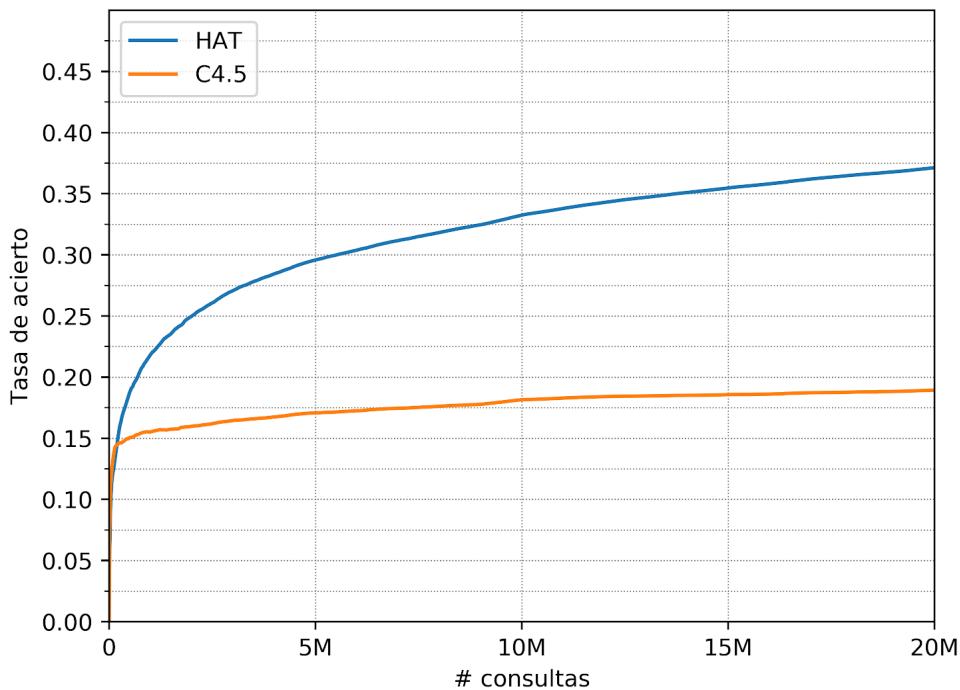


Figura 38: Comparación del rendimiento de usando C4.5 y HAT con tamaño de caché infinito.

Con el fin de analizar aquellos errores que provocan la importante diferencia entre la tasa de acierto de C4.5 y HAT, se realiza un conteo de consultas clasificadas como falsas negativas o falsas infrecuentes (Error de tipo 2). Para HAT, la cantidad de consultas comprendidas dentro de ese conjunto es de 1,6M, mientras que para C4.5 es de 5,1M. Es decir, C4.5 incurre en el error de tipo 2 en más de un cuarto del conjunto de consultas. Los errores de este tipo representan un

desaprovechamiento de la caché que impacta de forma desigual a los clasificadores. Esto se debe a que podría suceder que una consulta clasificada por HAT con un error de tipo 2, sea correctamente clasificada, luego de una adaptación del modelo. Sin embargo, al producirse la misma situación con C4.5, la consulta será erróneamente clasificada en todas sus ocurrencias, por lo menos hasta que el modelo sea reconstruido.

6. Conclusiones y trabajos futuros

En este trabajo se realiza una comparación del rendimiento de un árbol de decisión clásico (C4.5) contra el de un árbol de decisión adaptativo (HAT) en la tarea de predecir futuras ocurrencias de consultas en el entorno de un ambiente de búsqueda. En los experimentos, se utiliza para la evaluación un registro de consultas de un motor de búsqueda real, donde HAT muestra una considerable mejora en la efectividad de la caché gracias a su capacidad de adaptación a cambios de concepto.

En una primera etapa, en la que se evalúa la calidad de clasificación, HAT alcanza una exactitud superior a la de C4.5 con una diferencia porcentual que no resulta a priori demasiado significativa. Sin embargo, al efectuar experimentos relacionados con la gestión de las políticas de admisión, se manifiesta la superioridad mayor de HAT por sobre el otro algoritmo. Éste logra duplicar la tasa de aciertos de C4.5 para la mayoría de capacidades de caché empleadas. Bajo la misma configuración y solo con capacidades de caché que van desde 1K a 7,5K (las capacidades más reducidas), sería conveniente emplear a C4.5, sin embargo estos tamaños de caché no son reales para un sistema actual. Un análisis minucioso, empleando la tabla de contingencia, permite detectar que el árbol de decisión tradicional tiene, para esta tarea, un comportamiento conservador al asignar la clase ‘no singleton’, que lo hace clasificar un gran porcentaje de consultas frecuentes como falsas negativas.

Respecto de los cambios de concepto, se lleva a cabo un experimento para determinar la capacidad de HAT para adaptarse a cambios realmente abruptos. El algoritmo demuestra tener rápida respuesta, acondicionando el modelo para que se ajuste precisamente a la variación introducida artificialmente.

En la evaluación de la clasificación, HAT muestra un comportamiento que llama la atención. El modelo generado por el algoritmo presenta un crecimiento gradual en la tasa de error conforme más instancias son clasificadas. Si bien, con C4.5 sucede lo mismo y esta tendencia se encuentra más acentuada, esto es perfectamente explicable por la condición estática de su modelo. El sobreajuste del modelo de HAT es una explicación válida para este hecho. Utilizando un analizador sintáctico del modelo, desarrollado para esta tarea, se constata que el árbol generado por HAT alcanza una profundidad cercana a los treinta niveles. Por lo que podría

considerarse que una poda insuficiente ocasiona un deterioro gradual en la capacidad de generalización del modelo.

6.1. Trabajos futuros

Como trabajo futuro, se considera combinar la caché de llenado dinámico con un fragmento estático. En éste se almacenan los resultados de las consultas históricamente frecuentes. Esta estrategia híbrida de caché es considerada actualmente como la técnica de caché más efectiva para resultados en motores de búsqueda. Por otra parte, se planea incorporar atributos asociados al índice invertido para intentar mejorar la performance de la clasificación. Otra experimentación futura es la utilización de técnicas de selección de atributos, que tienen por objetivo la obtención del subconjunto de variables útiles para construir un buen predictor.

Con el objetivo de mejorar la política de admisión propuesta, se planea estudiar y determinar objetivamente la causa del crecimiento gradual en la tasa de error del modelo generado por HAT.

Además, se espera implementar la topología basada en Storm propuesta y ejecutar sobre ella diferentes pruebas asociadas a cuantificar su eficiencia y aceleración frente a la tarea para la que fue diseñada.

Finalmente, se propone evaluar el desempeño del algoritmo Vertical Hoeffding Tree (VHT), versión distribuida del Hoeffding Tree original, frente a las mismas tareas realizadas con HAT. VHT se ejecuta sobre la plataforma Apache Storm y aunque este algoritmo carece de mecanismos para la detección de cambios de concepto, al igual que la versión en la cual está basado, es un buen modelo a seguir para desarrollar una implementación distribuida de HAT. Contar con una versión distribuida puede resultar necesaria ante un mayor volumen de consultas.

7. Bibliografía

1. Shih B-Y, Chen C-Y, Chen Z-S. Retracted: An Empirical Study of an Internet Marketing Strategy for Search Engine Optimization. *Human Factors and Ergonomics in Manufacturing & Service Industries*. 2012;23: 528–540.
2. Baeza-Yates R, Gionis A, Junqueira FP, Murdock V, Plachouras V, Silvestri F. Design trade-offs for search engine caching. *ACM Transactions on the Web*. 2008;2: 1–28.
3. van den Bosch A, Bogers T, de Kunder M. Estimating search engine index size variability: a 9-year longitudinal study. *Scientometrics*. 2016;107: 839–856.
4. De Francisci Morales G. SAMOA: A platform for mining big data streams. 2013. pp. 777–778.
5. Sagiroglu S, Sinanc D. Big data: A review. 2013 International Conference on Collaboration Technologies and Systems (CTS). 2013. doi:10.1109/cts.2013.6567202
6. Baeza-Yates R, Gionis A, Junqueira F, Murdock V, Plachouras V, Silvestri F. The impact of caching on search engines. Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '07. 2007. doi:10.1145/1277741.1277775
7. Tolosa GH. Técnicas de caching de intersecciones en motores de búsqueda. Doctor, Universidad Nacional de Buenos Aires. 2016.
8. Kucukyilmaz T, Barla Cambazoglu B, Aykanat C, Baeza-Yates R. A machine learning approach for result caching in web search engines. *Inf Process Manag*. 2017;53: 834–850.
9. Gan Q, Suel T. Improved techniques for result caching in web search engines. Proceedings of the 18th international conference on World wide web - WWW '09. 2009. doi:10.1145/1526709.1526768
10. Subasic I, Castillo C. The Effects of Query Bursts on Web Search. 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology. 2010. doi:10.1109/wi-iat.2010.59
11. Koza JR, Bennett FH, Andre D, Keane MA. Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming. *Artificial Intelligence in Design '96*. 1996. pp. 151–170.
12. Russell. *Artificial Intelligence: A Modern Approach*, 2/E. Pearson Education India; 2003.
13. Kotsiantis SB, Zaharakis ID, Pintelas PE. Machine learning: a review of classification and combining techniques. *Artificial Intelligence Review*. 2006;26: 159–190.
14. Fayyad UM. *Advances in Knowledge Discovery and Data Mining*. Mit Press; 1996.
15. Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From Data Mining to Knowledge Discovery in Databases. *AI Magazine*. 1996;17. doi:10.1609/aimag.v17i3.1230
16. Gama J. *Knowledge Discovery from Data Streams*. CRC Press; 2010.
17. Sayed-Mouchaweh M, Lugofer E. *Learning in Non-Stationary environments - Methods and Applications*. Springer New York Dordrecht Heidelberg London; 2012.

18. Domingos P, Hulten G. Mining high-speed data streams. Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '00. 2000.
doi:10.1145/347090.347107
19. Bifet, Albert and Holmes, Geoff and Kirkby, Richard and Pfahringer, Bernhard. Moa: Massive online analysis. *Journal of Machine Learning Research*. 2010;
20. Domingos P, Hulten G. Mining high-speed data streams. Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '00. 2000.
doi:10.1145/347090.347107
21. Bifet A, Gavaldá R. Adaptive Parameter-free Learning from Evolving Data Streams. 2009;
22. Stonebraker M, Çetintemel U, Zdonik S. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*. 2005;34: 42–47.
23. Baeza-Yates R, Castillo C, Junqueira F, Plachouras V, Silvestri F. Challenges on Distributed Web Retrieval. 2007 IEEE 23rd International Conference on Data Engineering. 2007.
doi:10.1109/icde.2007.367846
24. Tatikonda S, Barla Cambazoglu B, Junqueira FP. Posting list intersection on multicore architectures. Proceedings of the 34th international ACM SIGIR conference on Research and development in Information - SIGIR '11. 2011. doi:10.1145/2009916.2010045
25. Long X, Suel T. Three-Level Caching for Efficient Query Processing in Large Web Search Engines. *World Wide Web J Biol*. 2006;9: 369–395.
26. Tolosa GH. Técnicas de caching de intersecciones en motores de búsqueda. Doctor, Universidad Nacional de Buenos Aires. 2016.
27. Baeza-Yates R, Gionis A, Junqueira F, Murdock V, Plachouras V, Silvestri F. The impact of caching on search engines. Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '07. 2007.
doi:10.1145/1277741.1277775
28. Baker M. Cluster computing white paper. arXiv preprint cs/0004014. 2000;
29. Kucukyilmaz T, Barla Cambazoglu B, Aykanat C, Baeza-Yates R. A machine learning approach for result caching in web search engines. *Inf Process Manag*. 2017;53: 834–850.
30. Alici SAA, Ozcan ISA, "O. Adaptive time-to-live strategies for query result caching in web search engines. European Conference on Information Retrieval; 2012.
31. Rifat O, Ismail SA, Barla Cambazoglu B, Ozgur U. Second Chance: A Hybrid Approach for Dynamic Result Caching and Prefetching in Search Engines. *ACM Transactions on the Web (TWEB)*. 2013;
32. Lempel R, Moran S. Predictive caching and prefetching of query results in search engines. Proceedings of the twelfth international conference on World Wide Web - WWW '03. 2003.
doi:10.1145/775152.775156
33. Gan Q, Suel T. Improved techniques for result caching in web search engines. Proceedings of the 18th international conference on World wide web - WWW '09. 2009. doi:10.1145/1526709.1526768
34. Ozcan R, Altingovde IS, Ulusoy Ö. Cost-Aware Strategies for Query Result Caching in Web Search Engines. *ACM Transactions on the Web*. 2011;5: 1–25.

35. Feuerstein E, Tolosa G. Cost-aware Intersection Caching and Processing Strategies for In-memory Inverted Indexes. Proc of 11th Workshop on Large-scale and Distributed Systems for Information Retrieval, LSDS-IR. 2014;
36. Baeza-Yates R, Gionis A, Junqueira FP, Murdock V, Plachouras V, Silvestri F. Design trade-offs for search engine caching. ACM Transactions on the Web. 2008;2: 1–28.
37. Tolosa G, Becchetti L, Feuerstein E, Marchetti-Spaccamela A. Performance Improvements for Search Systems Using an Integrated Cache of Lists Intersections. Lecture Notes in Computer Science. 2014. pp. 227–235.
38. Tolosa G. Técnicas de caching de intersecciones en motores de búsqueda. Phd. 2016.
39. Tolosa G, Becchetti L, Feuerstein E, Marchetti-Spaccamela A. Performance Improvements for Search Systems Using an Integrated Cache of Lists Intersections. Lecture Notes in Computer Science. 2014. pp. 227–235.
40. Breiman L. Random Forests. Springer. 2001;
41. Pass G, Chowdhury A, Torgeson C. A picture of search. Proceedings of the 1st international conference on Scalable information systems - InfoScale '06. 2006. doi:10.1145/1146847.1146848
42. Kucukyilmaz T, Barla Cambazoglu B, Aykanat C, Baeza-Yates R. A machine learning approach for result caching in web search engines. Inf Process Manag. 2017;53: 834–850.
43. Kucukyilmaz T, Barla Cambazoglu B, Aykanat C, Baeza-Yates R. A machine learning approach for result caching in web search engines. Inf Process Manag. 2017;53: 834–850.
44. Gama J. Knowledge Discovery from Data Streams. CRC Press; 2010.
45. Hierons R. Machine learning. Tom M. Mitchell. Published by McGraw-Hill, Maidenhead, U.K., 1997. ISBN: 0-07-115467-1, 414 pages. Softw Test Verif Reliab. 1999;9: 191–193.
46. Breiman L, Friedman J, Stone CJ, Olshen RA. Classification and Regression Trees. Chapman and Hall/CRC; 1984.
47. Quinlan JR. Induction of decision trees. Mach Learn. 1986;1: 81–106.
48. Ryzin JV, Van Ryzin J, Breiman L, Friedman JH, Olshen RA, Stone CJ. Classification and Regression Trees. J Am Stat Assoc. 1986;81: 253.
49. Quinlan JR. C45. 1993. pp. 1–16.
50. Hssina B, Merbouha A, Ezzikouri H, Erritali M. A comparative study of decision tree ID3 and C4.5. International Journal of Advanced Computer Science and Applications. 2014;4. doi:10.14569/specialissue.2014.040203
51. Domingos P, Hulten G. Mining high-speed data streams. Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '00. 2000. doi:10.1145/347090.347107
52. Fagni T, Perego R, Silvestri F, Orlando S. Boosting the performance of Web search engines. ACM Transactions on Information Systems. 2006;24: 51–78.
53. Pfahringer B, Holmes G, Kirkby R. Handling Numeric Attributes in Hoeffding Trees. Lecture Notes in Computer Science. pp. 296–307.

54. Greenwald M, Khanna S. Space-efficient online computation of quantile summaries. ACM SIGMOD Record. 2001;30: 58–66.
55. Hulten G, Spencer L, Domingos P. Mining time-changing data streams. Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '01. 2001. doi:10.1145/502512.502529
56. Wang H, Fan W, Yu PS, Han J. Mining concept-drifting data streams using ensemble classifiers. Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '03. 2003. doi:10.1145/956755.956778
57. Bifet A. Adaptive Stream Mining: Pattern Learning and Mining from Evolving Data Streams. IOS Press; 2010.
58. Bifet A, Gavaldà R. Learning from Time-Changing Data with Adaptive Windowing. Proceedings of the 2007 SIAM International Conference on Data Mining. 2007. pp. 443–448.
59. Bifet A, Gavaldà R. Learning from Time-Changing Data with Adaptive Windowing. Proceedings of the 2007 SIAM International Conference on Data Mining. 2007. pp. 443–448.
60. Datar M, Gionis A, Indyk P, Motwani R. Maintaining Stream Statistics over Sliding Windows. SIAM J Comput. 2002;31: 1794–1813.
61. Stonebraker M, Çetintemel U, Zdonik S. The 8 requirements of real-time stream processing. ACM SIGMOD Record. 2005;34: 42–47.
62. Ranjan R. Streaming Big Data Processing in Datacenter Clouds. IEEE Cloud Computing. 2014;1: 78–83.
63. Pass, Greg and Chowdhury, Abdur and Torgeson, Cayley. A picture of search. InfoScale. 152.
64. Qing C, Mu L, Ming Z. Improving Query Spelling Correction Using Web Search Results. Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning. 2007; 180–189.
65. Cucerzan S, Brill E. Spelling correction as an iterative process that exploits the collective knowledge of web users. Microsoft Research One Microsoft Way. 2004;
66. Behrens JT. Principles and procedures of exploratory data analysis. Psychol Methods. 1997;2: 131–160.
67. Baeza-Yates R, Gionis A, Junqueira F, Murdock V, Plachouras V, Silvestri F. The impact of caching on search engines. Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '07. 2007. doi:10.1145/1277741.1277775
68. Silverstein C, Marais H, Henzinger M, Moricz M. Analysis of a very large web search engine query log. ACM SIGIR Forum. 1999;33: 6–12.
69. Ahmad F, Kondrak G. Learning a spelling error model from search query logs. Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing - HLT '05. 2005. doi:10.3115/1220575.1220695

70. Sun, Xu and Gao, Jianfeng and Micol, Daniel and Quirk, Chris. Learning phrase-based spelling error models from clickthrough data. Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics. 2010; 266–274.
71. Tolosa GH. Técnicas de caching de intersecciones en motores de búsqueda. Doctor, Universidad Nacional de Buenos Aires. 2016.
72. Baeza-Yates R, Gionis A, Junqueira F, Murdock V, Plachouras V, Silvestri F. The impact of caching on search engines. Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '07. 2007. doi:10.1145/1277741.1277775
73. Charikar M, Chen K, Farach-Colton M. Finding Frequent Items in Data Streams. Lecture Notes in Computer Science. 2002. pp. 693–703.
74. Cormode G, Muthukrishnan S. An improved data stream summary: the count-min sketch and its applications. J Algorithm Comput Technol. 2005;55: 58–75.
75. Rabl T, Gómez-Villamor S, Sadoghi M, Muntés-Mulero V, Jacobsen H-A, Mankovskii S. Solving big data challenges for enterprise application performance management. Proceedings VLDB Endowment. 2012;5: 1724–1735.
76. Fawcett T. An introduction to ROC analysis. Pattern Recognit Lett. 2006;27: 861–874.
77. Croft B, Metzler D, Strohman T. Search Engines: Information Retrieval in Practice. Pearson Higher Ed; 2011.
78. Witten IH, University of Waikato. Department of Computer Science. Weka: Practical Machine Learning Tools and Techniques with Java Implementations. 1999.
79. Hastie T, Tibshirani R, Friedman J. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Science & Business Media; 2013.
80. Witten, Ian H. Frank, Eibe Hall, Mark A. Pal, Christopher J. Data Mining: Practical Machine Learning Tools and Techniques. 4th ed. 50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States: Elsevier; 2017.
81. Bifet A, Holmes G, Pfahringer B, Read J, Kranen P, Kremer H, et al. MOA: A Real-Time Analytics Open Source Framework. Lecture Notes in Computer Science. 2011. pp. 617–620.
82. Gama J, Sebastião R, Rodrigues PP. Issues in evaluation of stream learning algorithms. Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '09. 2009. doi:10.1145/1557019.1557060
83. De Moor Bart CM. Hyperparameter Search in Machine Learning. arXiv preprint arXiv:150202127. 2015;
84. Gama J, Gaber MM. Learning from Data Streams: Processing Techniques in Sensor Networks. Springer Science & Business Media; 2007.
85. Tsymbal A. The problem of concept drift: definitions and related work. Computer Science Department, Trinity College Dublin. 2004;106.
86. Bifet A, Gavaldá R. Adaptive Parameter-free Learning from Evolving Data Streams. 2009;
87. Podlipnig S, Böszörmenyi L. A survey of Web cache replacement strategies. ACM Computing Surveys. 2003;35: 374–398.

88. Zhang J, Long X, Suel T. Performance of compressed inverted list caching in search engines. Proceeding of the 17th international conference on World Wide Web - WWW '08. 2008. doi:10.1145/1367497.1367550
89. Kucukyilmaz T, Barla Cambazoglu B, Aykanat C, Baeza-Yates R. A machine learning approach for result caching in web search engines. *Inf Process Manag.* 2017;53: 834–850.