UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

TEXT MINING APLICADO A DOCUMENTACIÓN DE API PARA LA DETECCIÓN DE DIRECTIVAS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

GABRIEL ANDRÉS JOSUÉ CORREA GAETE

PROFESOR GUÍA:
ROMAIN ROBBES

MIEMBROS DE LA COMISIÓN:
JORGE PÉREZ ROJAS.
BÁRBARA POBLETE LABRA.

SANTIAGO DE CHILE
2015

Resumen

En esta memoria de Título se estudia la factibilidad de detectar directivas de API usando herramientas de Machine Learning y Text Mining. Las directivas son instrucciones importantes sobre el correcto uso de una API junto con restricciones y precauciones para prevenir errores. Sin embargo, existe la necesidad de destacar las directivas ya que pueden pasar desapercibidas para los desarrolladores de software. El objetivo general de este trabajo es diseñar e implementar un sistema de detección semi-automático de directivas de API. En este proyecto se realiza una búsqueda de los mejores algoritmos de Machine Learning para detectar las directivas de una API.

Se realizaron una serie de experimentos de Text Mining para evaluar la precisión de algoritmos que intentan separar las frases de una documentación en *directivas* y en *no-directivas*. Previo a la realización de los experimentos, fue necesario reunir una colección de *directivas* y de *no-directivas*, requeridos para 'entrenar' los programas de Machine Learning. Para facilitar la recopilación de estos datos, se implementó *Comments Highlighter* o *CHi*: una aplicación web que ayuda a buscar y destacar manualmente las directivas de una API. De este modo, se utilizó la herramienta implementada para reunir los datos precisados por los algoritmos de clasificación y posteriormente se realizaron pruebas para medir el rendimiento de la detección automática de directivas. Luego, la habilidad de detectar las directivas es agregada a la aplicación, y como la detección no es perfecta, los errores pueden ser corregidos manualmente usando *CHi*. Es por esto que el sistema es denominado como una solución semi-automática.

Los resultados demuestran que es factible detectar directivas usando clasificadores de *Machine Learning*. Además, se hacen pruebas variando el tamaño de los datos usados para entrenar los clasificadores, obteniendo información sobre cuántas muestras es necesario reunir para lograr un porcentaje satisfactorio de directivas detectadas. Finalmente se observa que hay dos algoritmos que funcionan significativamente mejor que los otros y uno de ellos es agregado a las funcionalidades de *CHi*.

Tabla de Contenido

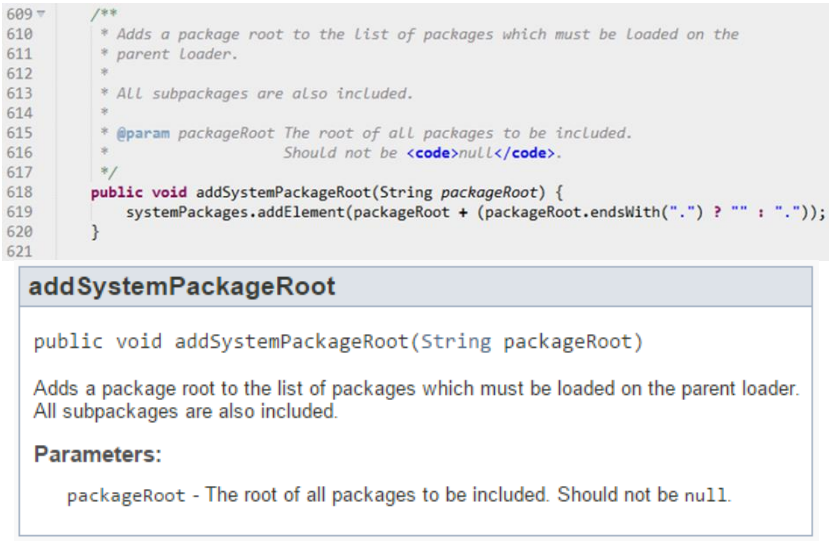
1 Introducción

1.1 Documentación de API

En la actualidad, existe una amplia variedad de programas y herramientas disponibles en el mercado de software. Estos programas son utilizados por distintos usuarios que deben conocer cómo manejar tales herramientas, lo cual normalmente significa pasar por una etapa de aprendizaje para comenzar el uso de un software. En particular existen miles de programas creados para ser usados por desarrolladores de software. Estos programas son llamados librerías o API[1] y del mismo modo que otros programas deben ser aprendidos a usar correctamente por sus usuarios.

Sistemas de software modernos suelen hacer uso de múltiples herramientas externas y APIs. Esto les ayuda a crear mejores programas o acortar los tiempos de desarrollo. Pero herramientas nuevas de este tipo son creadas todo el tiempo y es común que desarrolladores tengan que actualizar sus conocimientos aprendiendo a usar librerías nuevas para mantenerse en la competencia del mercado o cumplir con los requisitos de algunos empleos. Además, estas herramientas son frecuentemente modificadas y actualizadas con nuevas funcionalidades, intensificando la necesidad de que los programadores mantengan su conocimiento al día. Por esta razón, los manuales que explican el uso de una API son frecuentemente visitados por los desarrolladores de software. Estos manuales son llamados documentación de API y son generalmente provistos en la web del creador de cada librería. [1]

Los autores de librerías suelen invertir esfuerzos considerables en la creación de documentación donde especifican con detalle todo lo que los clientes necesitan saber sobre el correcto uso de su API. Las instrucciones de uso son usualmente escritas en los comentarios que acompañan al código fuente y en el caso del lenguaje Java por ejemplo, los comentarios de estilo *Javadoc* son ampliamente usados para documentar. La Imagen 1.1 muestra un ejemplo de documentación de la librería *Ant* de Java donde se escribió un texto explicativo antes del código de una función.



```
609    /**
610     * Adds a package root to the list of packages which must be loaded on the
611     * parent loader.
612     *
613     * All subpackages are also included.
614     *
615     * @param packageRoot The root of all packages to be included.
616     *                    Should not be <code>null</code>.
617     */
618    public void addSystemPackageRoot(String packageRoot) {
619        systemPackages.addElement(packageRoot + (packageRoot.endsWith(".") ? "" : "."));
620    }
621
```

addSystemPackageRoot

```
public void addSystemPackageRoot(String packageRoot)
```

Adds a package root to the list of packages which must be loaded on the parent loader. All subpackages are also included.

Parameters:

`packageRoot` - The root of all packages to be included. Should not be `null`.

Imagen 1.1: Ejemplo de documentación en Java de tipo Javadoc. Arriba aparece el comentario adentro del archivo de código y abajo se encuentra el mismo comentario en forma de HTML después de ser procesado por Javadoc.

[1] API: Application Programming Interface es un conjunto de funciones y herramientas que pueden ser usadas por programadores al crear software, apuntando a ahorrar trabajo o mejorar la calidad del código.

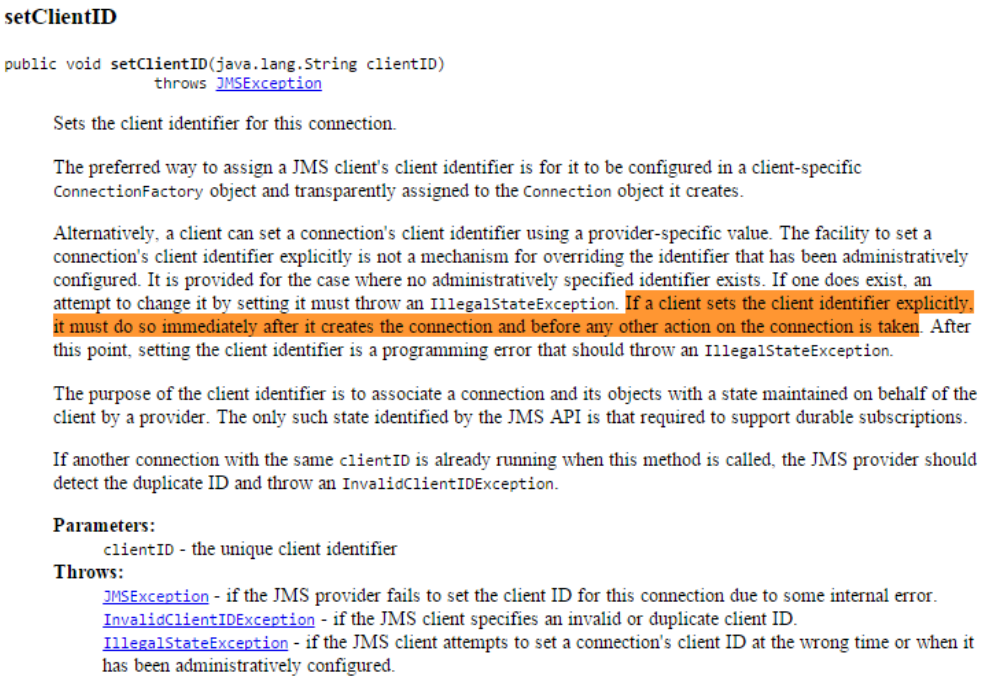
1.2 Problema con las Directivas

En todo manual de instrucciones pueden existir sugerencias críticas, o al menos sugerencias más importantes que otras, como por ejemplo la instrucción de no encender un microondas con algún metal en su interior por razones de seguridad. Así mismo, en documentación de API hay instrucciones críticas o sugerencias importantes las cuales llamamos *directivas* y no percatarse de ellas puede ocasionar errores difíciles de solucionar posteriormente. La siguiente instrucción es un ejemplo de directiva: *'Se debe llamar a la función IniciarSecuencia antes de modificar esta variable. El no hacerlo puede causar resultados inesperados'*. La siguiente definición de directiva se considera para este trabajo:

> *Directiva:*
>
> > *Las directivas de API son declaraciones en lenguaje natural que permiten a los desarrolladores ser conscientes de las instrucciones y restricciones relacionadas con el uso de una API.*

Como se mencionó, las directivas pueden pasar desapercibidas para un desarrollador que está revisando la documentación. Algunas funcionalidades de API son documentadas en extensos párrafos donde las directivas no se encuentran destacadas de ninguna forma. Así lo muestra la Imagen 1.2 (imagen obtenida de [2]), donde hay una instrucción importante al medio de un largo párrafo y no se encuentra destacada de modo alguno (la directiva fue destacada en la imagen para ejemplificar y en la documentación original no se encontraba destacada). Todo esto se traduce en un probable aumento de errores en el programa desarrollado, más tiempo solucionando *bugs* y más problemas de mantenimiento del código. Los resultados de un estudio [3], revisado con mayor detalle en la sección 2.1.2, apoyan la utilidad de destacar directivas en la documentación y a ayudar a que los desarrolladores puedan advertir las instrucciones importantes.

Este problema se presenta en mayor manera para un desarrollador con prisa, o que no tiene el tiempo suficiente para conocer a fondo todas las librerías externas que puede estar usando. Conocer todas las especificaciones de uso de una API completa puede ser una tarea muy complicada. Una librería típica puede tener cientos o miles de funciones, cada una con sus instrucciones de uso y explicación. Además existen dependencias entre funciones de una API, estados internos de una API y restricciones específicas sobre qué acciones evitar al usar una librería. Puede ocurrir que un desarrollador no realice una lectura completa de la documentación, más bien prefiriendo revisarla cuando la necesite para comprender funcionalidades puntuales. Esto es referido de la forma *'explore documentation with an "as-needed" strategy'* por Soloway et al. [4]



setClientID

```
public void setClientID(java.lang.String clientID)
                throws JMSException
```

Sets the client identifier for this connection.

The preferred way to assign a JMS client's client identifier is for it to be configured in a client-specific `ConnectionFactory` object and transparently assigned to the `Connection` object it creates.

Alternatively, a client can set a connection's client identifier using a provider-specific value. The facility to set a connection's client identifier explicitly is not a mechanism for overriding the identifier that has been administratively configured. It is provided for the case where no administratively specified identifier exists. If one does exist, an attempt to change it by setting it must throw an `IllegalStateException`. If a client sets the client identifier explicitly, it must do so immediately after it creates the connection and before any other action on the connection is taken. After this point, setting the client identifier is a programming error that should throw an `IllegalStateException`.

The purpose of the client identifier is to associate a connection and its objects with a state maintained on behalf of the client by a provider. The only such state identified by the JMS API is that required to support durable subscriptions.

If another connection with the same `clientID` is already running when this method is called, the JMS provider should detect the duplicate ID and throw an `InvalidClientIDException`.

Parameters:

`clientID` - the unique client identifier

Throws:

`JMSException` - if the JMS provider fails to set the client ID for this connection due to some internal error.
`InvalidClientIDException` - if the JMS client specifies an invalid or duplicate client ID.
`IllegalStateException` - if the JMS client attempts to set a connection's client ID at the wrong time or when it has been administratively configured.

Imagen 1.2: Documentación de un método de Java JMS. La frase destacada es una directiva que no es simple de detectar a primera vista pues se encuentra al medio de un largo párrafo.
(En la documentación original la directiva no aparece destacada)

1.3 Una Solución

Si las documentaciones de API tuvieran a las directivas destacadas de alguna manera, como por ejemplo en la Imagen 1.2 donde se destacó el color de fondo de una frase, los desarrolladores podrían percatarse con facilidad de instrucciones importantes. Para ayudar a que las directivas sean más fácilmente detectables se podría seguir un procedimiento estandarizado al crear documentación de API, destacando las directivas manualmente al escribir los comentarios del código fuente. Pero esta opción probablemente tardaría años en adoptarse y puede ser complicado lograr que sea utilizado en todos los proyectos de software. Otra forma, es destacar las directivas de una documentación una vez que ya se encuentra terminada, realizando algún análisis de los comentarios.

Para destacar las directivas primero es necesario detectar cuáles son las instrucciones importantes de la documentación y como esto tiene un componente subjetivo, se podría pensar que la mejor forma es que una o más personas destaquen manualmente cada directiva. Sin embargo una librería puede tener demasiados comentarios como para ser revisados manualmente por una persona. Por ejemplo, Java tiene más de 4000 clases, cada una con varios métodos comentados. Posiblemente se podría realizar una revisión con múltiples personas, cada uno con conocimientos en documentación y directivas, pero esto podría alcanzar gastos elevados contratando a tal grupo de expertos. Una revisión manual puede ser muy tediosa y además costosa en horas-hombre para una empresa.

El plugin eMoose es un plugin para Eclipse que ayuda a destacar directivas (ver Imagen 2.3 en la página 6). Sin embargo eMoose puede destacar directivas de pocas API (Java 6 y Eclipse 3.4 API). Para agregar nuevas librerías a eMoose es necesaria una revisión manual de ellas, para identificar las directivas.

Una posible solución – propuesta por este trabajo – es que se detecten de manera semi-automática las directivas de API, usando un clasificador entrenado con métodos de *Machine Learning* y herramientas de *Text Mining*, el cual puede aprender con comentarios de librerías. Es decir, un programa intentará adivinar la mayor cantidad de directivas posibles y luego, opcionalmente, se puede revisar de forma manual los resultados para corregir errores. La solución incluye el desarrollo de una aplicación web que permite extraer los comentarios de una API para luego visualizarlos y corregir errores de la clasificación automática, además de una búsqueda de un buen clasificador de Machine Learning que logre seleccionar la mayor cantidad de directivas que pueda.

Un programa puede aprender a detectar directivas, la duda es ¿con cuánta eficacia o precisión lo puede hacer? El rendimiento de estos clasificadores es medido en el presente trabajo a lo largo de una serie de experimentos aplicando Text Mining.

1.4 Objetivos

1.4.1 Objetivos Generales

El objetivo general de este trabajo es diseñar e implementar un sistema de detección semi-automático de directivas de API basado en la aplicación de Text Mining. Junto con esto, realizar una búsqueda de los mejores algoritmos de Machine Learning para detectar directivas.

1.4.2 Objetivos Específicos

Los objetivos específicos son los siguientes:

- Diseñar e implementar una aplicación que permita destacar manualmente las directivas de los comentarios de una API.
- Realizar una revisión manual de comentarios de API para obtener datos con los cuales poder aplicar Text Mining
- Investigar la capacidad que tienen algunos algoritmos de Machine Learning para detectar automáticamente las directivas en los comentarios de API.
- Agregar a la aplicación implementada que destaca directivas un componente de detección automática de directivas de API.

2 Marco teórico y trabajos anteriores

2.1 Destacación de directivas en API

2.1.1 Syntax highlighting en editores de texto

Para destacar directivas de API hay que considerar que hay principalmente dos lugares donde la documentación es revisada: en el código fuente y por otra parte en páginas web donde suele publicarse una documentación oficial.

En particular, si es necesario destacar directivas sobre el código fuente existe la posibilidad de usar el coloreo de sintaxis que ofrecen algunos editores de texto. El coloreo de sintaxis es un mecanismo común que usan los editores de texto para darle formato a algunas partes del código, como color de fuente o color de fondo. Lo más común es usarlo para colorear el texto de variables, funciones y palabras reservadas del lenguaje de programación, cada uno con diferentes colores. El mismo mecanismo, el cual no modifica de ningún modo el código fuente del archivo original, puede permitir destacar directivas.

La Imagen 2.1 muestra una prueba de concepto realizada durante este trabajo en el editor de texto Sublime Text, hecha con el objetivo de comprobar si es factible destacar directivas usando coloreo de sintaxis. El editor de texto quedó programado para destacar una línea completa cuando encontrase la expresión *"@tag usage.restriction"* adentro de un comentario – el comentario que comienza con "@tag" proviene del uso de *TagSea*[2], un plugin de Eclipse que permite marcar puntos de interés dentro del código fuente y es usado a modo de ejemplo. De esta forma, si se marcan las directivas con "*@tag directive"*, podrían aparecer destacadas.

De manera similar, en la Imagen 2.2 se destacan las palabras *'Must'* cuando se encuentran adentro de un comentario. Una frase con la palabra *'must'* tiene una mayor probabilidad de ser una instrucción importante en muchos casos, y por esta razón existe la estrategia de detectar directivas automáticamente considerando un conjunto de palabras clave (o *keywords*). Esto se explica en mayor detalle en la sección 2.2 y se estudia su precisión en la sección 6.2.2. De todos modos, se puede ver que es posible destacar palabras claves dentro de los editores de texto, lo cual podría ser una forma de destacar posibles directivas.

De hecho, como una experimento adicional, se observó que es posible tener comentarios que comienzan con '*//*' de un color, y comentarios que comienzan con '*///*' de otro color. Esto, que se puede ver en ambas imágenes, podría permitir escribir directivas durante la creación de código, marcando los comentarios que son directivas con un color distinto al de los otros comentarios. Incluso podría servir para marcar los comentarios *'TODO'*, los cuales indican que hay algún trabajo pendiente, con un color especial.

[2] TagSea: http://tagsea.sourceforge.net/

```
// Función factorial: Recibe un int mayor o igual a 0.
//                    Retorna el factorial del número entregado
// @tag usage.restriction : Must be called as a java applet, not as a java application.
function factorial(n) {
    /// caso basse: factorial de 0 es igual a 1
    if (n === 0) {
        return 1;
    }

    /// fac(n) = n*fac(n-1)
    return n * factorial(n - 1);
}
```

Imagen 2.1: Línea completa siendo destacada gracias al coloreo de sintaxis dentro de Sublime Text, un editor de texto.

```
// Función factorial: Recibe un int mayor o igual a 0.
//                    Retorna el factorial del número entregado
// @tag usage.restriction : Must be called as a java applet, not as a java application.
function factorial(n) {
    /// caso basse: factorial de 0 es igual a 1
    if (n === 0) {
        return 1;
    }

    /// fac(n) = n*fac(n-1)
    return n * factorial(n - 1);
}
```

Imagen 2.2: Palabra 'Must' siendo destacada gracias al coloreo de sintaxis dentro de Sublime Text, un editor de texto.

Para lograr todo esto, fue necesario extender el módulo de coloreo de sintaxis de Sublime Text, pero una vez implementado, se puede compartir la funcionalidad creando un plugin que queda disponible a ser usado por otros usuarios. Varios editores de texto que soportarían el destacar directivas usando coloreo de sintaxis. De hecho otro ejemplo es dado en la siguiente sección; un plugin de la IDE Eclipse llamado eMoose que permite destacar las directivas en código de Java.

2.1.2 eMoose

El proyecto *eMoose* se trata de un plugin del programa Eclipse que ayuda a destacar las directivas de la documentación[3]. La Imagen 2.3 muestra dos métodos destacados por eMoose por contener directivas en sus documentaciones de API. De este modo, un usuario puede percatarse de los elementos de una API que contienen instrucciones importantes de uso. Además se puede ver en la Imagen 2.4 una ventana de ayuda de Eclipse donde las tres últimas líneas fueron agregadas por eMoose. Estas indican que la documentación del método *setClientID(String)* contiene tres directivas.

Las ventanas de ayuda en Eclipse se muestran al mantener el cursor sobre métodos y clases, haciendo mucho más fácil que un desarrollador se percate de las directivas relacionadas con su código. Pero un problema de eMoose es que tiene pocas librerías con la información requerida de las directivas. Las API que no han sido complementadas con la información de directivas, son mostradas en Eclipse sin ningún tipo de destacaciones que realiza eMoose. Esto puede suceder ya que la tarea de agregar las directivas de forma manual toma tiempo pues las librerías pueden contener cientos o miles de funciones.

[3] Sitio oficial de eMoose: https://code.google.com/p/emoose-cmu/

Un sistema que ayude a identificar las directivas de forma automática o semi-automática podría complementarse bien con eMoose, proveyendo de la información de las librerías que eMoose necesita para presentar su ayuda con diversas API.

```
queueConnectionFactory = new ActiveMQConnectionFactory(BROKER_ADDRESS);
queueConnection = queueConnectionFactory.createQueueConnection();
queueSession = queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
queue = queueSession.createQueue(queueName);
```

Imagen 2.3: Ejemplo de código visualizado en Eclipse con eMoose

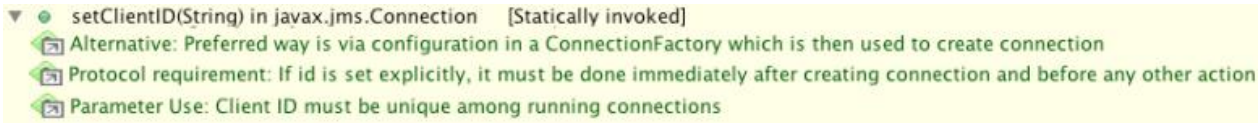


Imagen 2.4: Ejemplo de ventana emergente de Eclipse mostrando directivas agregadas por eMoose.

Junto con la implementación de eMoose, los creadores (*Dekel et al.* [5]) realizaron un estudio con el propósito de estudiar si los usuarios de la librería pueden percatarse de instrucciones importantes en la documentación con una mayor facilidad. Comparan el desempeño entre los que usan las decoraciones sobre el código y los que no las usan, considerando a 25 sujetos repartidos aleatoriamente entre el grupo experimental (EXP), el cual usa eMoose, y el grupo de control (CTL) que no usa eMoose. Los resultados indican que los sujetos tuvieron un mayor éxito detectando y corrigiendo errores de código al usar eMoose (ver Imagen 2.5). El estudio observó un promedio de éxito de 92% para los que usaron eMoose y un promedio de éxito de 41% para los que no lo usaron. Los resultados reafirman la importancia de las directivas en documentación de API.

Imagen 2.5: Tasa de éxito de desarrolladores corrigiendo bugs sin eMoose versus los que sí usaron eMoose. EXP corresponde a los que usaron eMoose; CTL, a los que no. (Verde representa éxito y rosado representa fracaso)

Además de resultados que obtuvieron, Dekel et al. presentan una lista de los tipos de directivas más prominentes que se encontraron durante una revisión de documentación de API que realizaron. La lista de tipos de directivas que crearon se encuentra en la Tabla 2.1.

Tabla 2.1: Tipos de directivas identificadas en el estudio de Dekel et al.

Restricciones
Protocolos
Locking
Parámetros y valores de retorno
Alternativas
Limitaciones
Efectos secundarios
Desempeño o Rendimiento
Threading
Seguridad

2.2 Trabajo de Monperrus et al.

Monperrus et al. [6] realizaron un estudio en el cual establecieron una clasificación de directivas. Por observación notaron que las directivas solían contener algunas palabras clave que se repetían como pasa en el siguiente ejemplo con la palabra 'encouraged'.

> While Deque implementations are not strictly required to prohibit the insertion of null elements, they are strongly encouraged to do so.[4]

> Subclasses of ClassLoader are encouraged to override {@link #findClass(String)}, rather than this method.[5]

Juntando todas estas palabras claves, a las cuales nos referiremos como *keywords*, ellos las usaron para extraer de documentación de API a todos los comentarios que contuvieran al menos uno de los *keywords*. Una vez obtenidos estos comentarios, verificaron manualmente si cada uno era efectivamente una directiva. Mientras revisaban los datos, fueron separando las directivas en tipos y subtipos, formando una taxonomía de directivas. Además juntaron estadísticas de cuantas directivas correctas podía adivinar cada *keyword*, ya que por ejemplo la palabra *'should'* juntó más comentarios que resultaron ser realmente directivas que la palabra *'extend*'*. Una parte de sus resultados se encuentran en la Imagen 2.6.

[4] Obtenido de java/util/Deque
[5] Obtenido de java/lang/ClassLoader

Table 3 The ability of syntactic patterns to reveal directives

Concern	Java	JFace	commons.collections
must	78.6±0.3% (740)	94.4±5.4% (179)	98.2±0.3% (330)
mandat*	42.9±0.0% (7)	– (0)	25.0±0.0% (4)
require*	70.3±3.4% (232)	58.3±6.2% (72)	29.6±6.9% (27)
should	67.5±0.3% (750)	83.9±5.7% (205)	76.4±5.6% (55)
encourage*	76.2±8.7% (21)	100.0±0.0% (6)	100.0±0.0% (4)
recommend*	88.4±8.9% (43)	95.5±9.2% (22)	0.0±0.0% (1)
may	60.5±5.5% (258)	93.7±3.3% (351)	57.1±7.5% (84)
extend*	22.6±8.7% (62)	68.7±0.0% (163)	16.7±4.1% (24)
overrid*	90.8±6.3% (130)	85.1±0.7% (249)	94.4±0.0% (54)

Imagen 2.6: (Parte de los datos de la Tabla 3 del trabajo de Monperrus et al. [6]) Porcentajes de éxito que tiene cada patrón sintáctico para revelar directivas de los comentarios de Java SDK, JFace y Apache Commons Collections. Los números entre paréntesis representan el número de comentarios analizados y los errores corresponden a un 95% de nivel de confianza. Un guión significa que el patrón sintáctico no fue hallado. Un asterisco significa que una palabra puede terminar en otras posibles terminaciones de la palabra (extend*: extend, extended, extender)

La taxonomía que crearon, ayuda a comprender un poco más la naturaleza de las directivas. Los tipos de directivas que distinguieron se encuentran en la Tabla 2.2. Además, en su estudio muestran ejemplos para cada tipo de directiva y definen las características que debe tener una frase para ser clasificada según corresponda. En la sección 4 se explica el proceso de revisión manual de comentarios que fue llevada a cabo en el presente estudio. Durante esta revisión fue fundamental adherirse en lo posible a las mismas reglas utilizadas por Monperrus et al.

Tabla 2.2: Taxonomía de directivas usada en el estudio de Monperrus et al. (traducidas al español)

- Directiva de Método:
  - No null
  - Valor de retorno
  - Visibilidad del método
  - Posibles excepciones
  - Null permitido
  - Formato de strings
  - Rango numérico
  - Tipo de parámetro de un método
  - Correlación entre parámetros de un método
  - Post llamada
  - Misceláneo
- Directiva de Subclases:
  - Redefinición de métodos[6]
  - Identificación de clase extensible
  - Implementación de método
  - Extensión de métodos
  - *Non-local consistency subclassing*
  - Contrato de llamada de un método
  - Misceláneo
- Directiva de Estado:
  - Secuencia de llamado de métodos
  - No basados en llamadas
- Alternativas
- Sincronización
- Misceláneo

[6] (Method overriding)

El estudio fue realizado sobre un subconjunto de tres APIs populares para Java: Java SDK (java.lang, java.util, java.io, java.math, java.net, java.rmi, java.sql, java.security, java.text, java.applet), JFace (parte del proyecto Eclipse) y Apache Commons Collections. Argumentan que estas APIs constituyen una muestra representativa y que son utilizadas por grandes organizaciones. En total trabajaron con una muestra de 2800 páginas de texto con un promedio de 500 palabras por página.

En este trabajo se estudiarán las mismas API consideradas por Monperrus et al. y se usará la lista de *keywords* para detectar directivas con el fin de tener un punto de comparación sobre la calidad de un programa para seleccionar las directivas.

3 Desarrollo de CHi

CHi es el programa implementado durante este trabajo. Fue inicialmente concebido como una manera de facilitar la visualización y revisión manual de comentarios de API, que permitiera destacar visualmente las frases que fueran directivas. La herramienta resultó ser necesaria para realizar una revisión manual de más de 3000 comentarios Javadoc. Los resultados de marcar las directivas fueron luego aplicados a Data Mining para obtener modelos que identifiquen automáticamente las directivas en datos futuros.

Finalmente, el desarrollo concluyó en un software que va a poder facilitar futuras revisiones semi-automáticas de documentación de API y que automatiza el pre-procesamiento necesario para poder aplicar algoritmos de Machine Learning en Weka a estos datos. El programa es una aplicación web, diseñado así para ser de fácil acceso para un usuario, requiriendo solo de un browser. También al ser una aplicación web, un browser permite visualizar el HTML que tiene un comentario javadoc, mejorando la visualización de la documentación.

El objetivo principal de CHi es facilitar la clasificación, manual o semi-automática, de frases en la documentación de las API así como también, permitir una fácil visualización y navegación de los comentarios. CHi ayuda a destacar directivas dentro de los comentarios de una API.

3.1 Usuarios Finales

Los usuarios finales son principalmente dos. A continuación se describen ambos usuarios a los cuales está destinada la aplicación.

3.1.1 Usuario Documentador de API

Este tipo de usuario final, es una persona que necesita documentar una librería con Javadoc y le interesa destacar las directivas de su API. Los usuarios que lean su documentación deben poder advertir con mayor facilidad las instrucciones importantes sobre el uso correcto de la librería. Entonces puede usar el programa CHi para que las directivas sean detectadas automáticamente. Luego si desea, puede revisar y corregir manualmente los errores de la selección de directivas, si es que los hay, usando la aplicación.

Este tipo de usuario puede documentar una librería grande con varias clases y comentarios. La revisión manual de todos los comentarios para identificar las directivas es un proceso demasiado largo, pero con CHi un alto porcentaje de las directivas son destacadas automáticamente. Luego el usuario puede revisar las frases destacadas, ya que algunas de ellas pudieron haber sido incorrectamente marcadas como directivas.

El usuario prefiere un programa que pueda identificar las directivas teniendo pocos errores. Un excelente caso para el usuario es cuando el programa identifica la gran mayoría de las directivas. En este caso, solo hay unas pocas directivas que no fueron destacadas, las cuales son muy difíciles de encontrar manualmente, pues hay que recorrer todos los comentarios de la API en el peor caso. Si son suficientemente pocas las directivas ignoradas, se cree que el usuario puede estar satisfecho con el porcentaje pequeño de errores, ya que al menos, así no debe revisar toda la API; sólo debe revisar las marcadas como directivas, donde pueden haber frases que no eran realmente directivas. Este trabajo estima que un usuario queda insatisfecho si menos de un 50% de las directivas fueron detectadas, que no está satisfecho aun cuando se alcanzan porcentajes entre 50% y 85% y que está si está satisfecho con un porcentaje mayor a 85% de las directivas detectadas. Estos números son solo especulaciones de lo que este tipo de usuario final puede esperar del programa y una forma de fijar una meta para el rendimiento del programa.

3.1.2 Usuario Investigador de Text Mining

Un investigador o desarrollador que desea aplicar Text Mining a la documentación de una API o bien a un conjunto de comentarios de código fuente. Posiblemente incluso crear mejores modelos de predicción de directivas. Va a usar el programa para pre-procesar datos que luego usará en Weka u otra herramienta similar. Puede además modificar manualmente cuáles son las directivas en la aplicación web para editar su set de datos con mayor facilidad.

El pre-procesamiento de datos para aplicar Text Mining puede ser tedioso. En el caso de texto de la documentación de una API, el pre-proceso de esta información significa extraer todos los comentarios de cada archivo de código fuente, posiblemente separarlo por frases y luego dejar los datos en un formato compatible con Weka por ejemplo. Es por esta razón que un usuario puede tener interés en usar las funciones de CHi.

3.2 Casos de Uso

3.2.1 Documentador de API

Usuario crea nuevo proyecto seleccionando una carpeta en su directorio local. Su carpeta contiene los archivos con extensión .java y contiene más carpetas internas que a su vez contienen otros archivos .java. La carpeta también contiene archivos .txt y .html los cuales no son considerados. Para terminar de crear el proyecto elige si el programa va a destacar las directivas o si el proyecto comenzará sin frases destacadas. Tiene tres opciones: no adivinar directivas, adivinar usando el clasificador de Machine Learning o adivinar usando keywords. El usuario elige adivinar automáticamente las directivas con un clasificador de Machine Learning que viene con la aplicación.

Luego, la página toma unos segundos en procesar los archivos y después aparece el primer comentario de API, indicando que el proyecto ya ha sido creado y está listo para ser manualmente revisado.

El usuario decide revisar una parte de las frases destacadas por el programa, usando las flechas y tecla de espacio de su teclado. Corrige algunas frases destacadas como directiva porque no eran instrucciones importantes y el programa se equivocó clasificando esos casos. Pero el usuario encuentra varios otros casos que si están correctamente adivinados por el programa. El usuario no revisa las frases no destacadas pues son demasiadas y sabe que es poco probable encontrar directivas que no fueron detectadas porque el clasificador de Machine learning de CHi suele equivocarse poco en esto.

3.2.2 Investigador de Text Mining

Usuario crea nuevo proyecto seleccionando una carpeta en su directorio local. Su carpeta contiene los archivos con extensión .java y contiene más carpetas internas que a su vez contienen otros archivos .java. La carpeta contiene otros archivos .txt y .html, pero estos no son considerados.

Luego de seleccionar la carpeta, la página toma unos segundos en procesar los archivos y luego muestra el primer comentario de API, indicando que el proyecto ya ha sido creado y está listo para ser manualmente revisado.

El usuario revisa y marca varias frases de la documentación de su API, usando las flechas y tecla de espacio de su teclado y cuando ha terminado, presiona el botón de exportar datos para generar un archivo .csv y un archivo .arff.

El usuario luego aplica algoritmos de Machine Learning o análisis de Text Mining en el programa Weka abriendo el archivo .arff que contiene las frases de documentación que él manualmente revisó. Ahora puede probar con distintos clasificadores en esos datos para mejorar su modelo predictivo de directivas, con el fin de obtener un clasificador que en el futuro pueda adivinar con más precisión cuales frases son directivas dentro de la documentación de una API.

3.3 Interfaz de CHi

En la siguiente imagen se muestra la interfaz de CHi y luego se detalla cada componente y funcionalidad.[8]

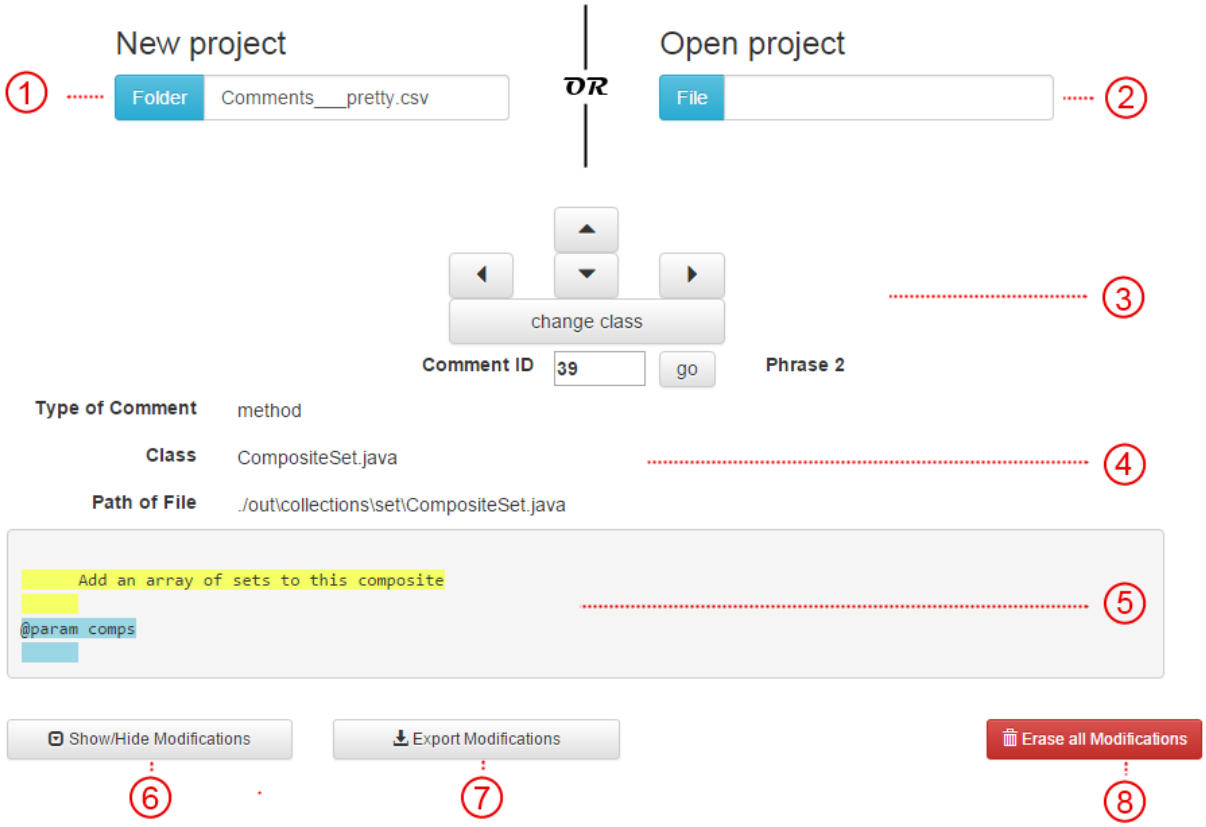


Imagen 3.1: Interfaz gráfica de CHi, indicando los botones y las funcionalidades de la aplicación.

1) Proyecto nuevo

Permite seleccionar una carpeta que contenga los archivos de código fuente de un proyecto Java para comenzar a destacar las directivas de la documentación.

2) Abrir proyecto

Permite continuar revisando un proyecto que ya existe. Debe ser un archivo csv.

[8] Esta versión aun no presentaba la funcionalidad de detectar directivas automáticamente.

3) Navegación

Permite navegar entre los comentarios y frases de la documentación, además de cambiar la clase de la frase seleccionada. Se pueden utilizar teclas de acceso rápido (ver Tabla 3.1). Además se puede ingresar un número de comentario para navegar directamente hacia él.

4) Información

Tipo de comentario (método, campo, clase, paquete), clase Java del comentario y ubicación del archivo que contiene al comentario.

5) Visualización de comentarios

Zona donde se muestran las frases de los comentarios. Las frases se pueden destacar para marcar el tipo o clase de frase que es (no-directiva, directiva, directiva-null, semi-directiva/por-revisar).

6) Mostrar/Ocultar frases modificadas

Ver lista de frases modificadas. Cada frase tiene una ID única que la identifica.

7) Exportar proyecto

Exportar la documentación de API del proyecto, considerando las modificaciones realizadas, a un archivo csv.

8) Borrar modificaciones

Borrar todas las modificaciones realizadas.

Además, la interfaz permite que un usuario utilice teclas de acceso rápido (*hotkeys*) para ofrecer una mejor usabilidad de la aplicación. La Tabla 3.1 a continuación muestra las teclas de acceso rápido y sus funciones.

Tabla 3.1: Teclas de acceso rápido (*hotkeys*) de la aplicación CHi.

Tecla	Alternativas		Función
Espacio			Cambiar el color de la frase seleccionada. Alterna entre los posibles colores al presionar varias veces.
Izquierda	(←)	J , A	Comentario anterior
Derecha	(→)	L , D	Comentario siguiente
Arriba	(↑)	I , W	Frase anterior
Abajo	(↓)	K , S	Frase siguiente

3.4 Implementación de CHi

La aplicación ***C****omments* ***Hi****ghlighter*, o bien 'CHi', fue implementada con el fin de facilitar el proceso de detectar y destacar, de manera semi-automática, las directivas de una API.

Los comentarios escritos junto al código fuente son los que conforman la documentación de una librería. En el caso de Java, los comentarios usualmente siguen la sintaxis pedida por Javadoc. Las documentaciones de API realizadas con Javadoc pueden visualizarse en una página web o bien en los comentarios que se encuentran dentro de archivos de código fuente.

A continuación se detallan las fases principales del desarrollo y detalles de su implementación.

3.4.1 Extracción de comentarios

El programa necesita poder acceder y revisar los comentarios de una API. Para esto necesita que el usuario seleccione la carpeta donde tiene los archivos de su proyecto Java. Luego extraerá todos los comentarios de tipo Javadoc que se encuentran al interior de cada archivo de extensión '.java', y los deja disponibles para ser visualizados en la página web.

Este proceso se realizó usando un script de Shell de unix llamado slocc.sh[9] el cual permite extraer los comentarios de varios archivos de manera recursiva por los directorios. Luego la salida de este script fue procesada por un script implementado en python que separa todos los comentarios en frases (usando expresiones regex) y luego junta todo en un archivo .csv, que puede ser entendido por la aplicación web y finalmente se muestran los comentarios en un browser.

3.4.2 Aplicación y hotkeys

La aplicación fue programada en HTML5, usando javascript, jQuery, Ajax y php. Además usa un archivo .css de Bootstrap para mejorar la presentación visual. Corre usando un modelo cliente-servidor donde el servidor es ejecutado por el lenguaje php de la forma *"php -S localhost:8080"*.

y el servidor es un programa que es parte de la api de php. Su uso sólo ha sido probado con éxito en Chrome versiones 42, 43 y 44. Se observaron errores en el sistema de almacenamiento local del cliente al usar Safari, por lo que se recomienda fuertemente usar la aplicación en Chrome.

La aplicación web permite el uso de hotkeys para navegar a través de los comentarios de API y para destacar frases. Se usan las teclas de navegación del teclado (arrows) para avanzar o retroceder y se usa la tecla de espacio para destacar la frase que aparece seleccionada en ese momento. Apretar la tecla de espacio nuevamente hace que la frase circule por los 4 tipos de comentarios: no-directiva (sin color de fondo), directiva (color amarillo de fondo), directiva-null (color de fondo rosado) y semi-directiva (color de fondo azul). La Imagen 3.2 muestra como alternan los colores al presionar la tecla de espacio. En uno de los experimentos de este trabajo, el color azul de la clase semi-directiva fue considerado como otra clase 'por-revisar', para indicar que era necesario discutir posteriormente si tales frases eran directivas o no-directivas (ver 4.1.1).

[9] http://vgoenka.tripod.com/unixscripts/slocc.sh.txt

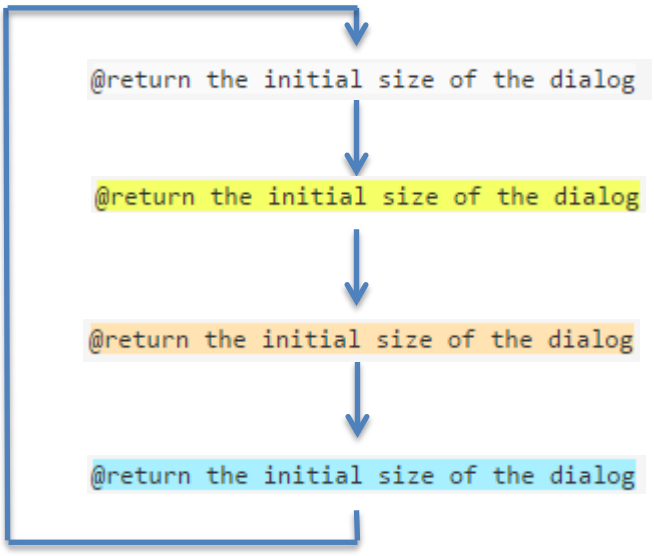


Imagen 3.2: Alternación de colores de las frases al presionar la tecla espacio en la aplicación CHi.

La aplicación almacena las correcciones realizadas por el usuario de forma local usando el método *LocalStorage* de HTML5. El sistema de destaque con colores de fondo es realizado usando css y jQuery.

Finalmente los datos pueden ser exportados a un archivo .csv el cual contiene los datos modificados por el usuario.

4 Revisión Manual de Comentarios de API

Fue necesario realizar una revisión manual de comentarios para obtener datos requeridos por las herramientas de Machine Learning, para poder aprender a clasificar comentarios de API. De hecho, se realizaron dos etapas de revisiones manuales la cuales ayudaron a formar los *datos preliminares* y los *datos finales*. Los datos preliminares no resultaron ser de la calidad esperada, lo cual dio razones para formar un refinado set de datos finales.

Cada comentario de API fue separado por las frases que lo componen. Durante la revisión de los datos preliminares, las frases fueron asignadas a la clase *no-directiva*, *directiva*, o a la clase *Por-revisar*, mientras que en los datos finales las frases fueron asignadas a la clase *no-directiva*, *directiva*, *semi-directiva* o a la clase *directiva-null*. Las clases *semi-directiva* y *directiva-null* se incluyeron para dejar la posibilidad abierta a usar esta distinción de clases de algún modo, pero finalmente no se usaron estas dos clases y fueron consideradas dentro de la clase *directiva*.

Todos los comentarios fueron extraídos del código fuente de JFace, Apache Commons Collections y Java, los cuales se extrajo un total de 71603 frases siendo Java la librería con mayor cantidad de frases (45270) y Apache Commons Collections la librería con menor cantidad de frases (7101). A lo largo del trabajo Apache Commons Collections será referido como Apache o Apache Commons.

La revisión manual de comentarios fue realizada usando CHi. El uso de la herramienta sin duda agilizó el proceso de revisión principalmente porque mejora la legibilidad de los comentarios y porque permite el uso de hotkeys (o teclas de acceso rápido) para navegar por la lista de comentarios y asignar clases a las frases.

La clasificación manual de comentarios no es un proceso objetivo, pues no existe una metodología ni pauta para separar los comentarios en clases. Aun si existiera una, es difícil imaginar que esta fuera objetiva y siempre correcta. Para mejorar la validez de los experimentos y resultados, la mayoría de los comentarios fueron revisados y clasificados por dos personas capacitadas en el asunto de directivas y documentación de API. Cada comentario fue leído y asignado por los revisadores de forma separada e independiente y los resultados fueron posteriormente comparados. Las diferencias entre los resultados fueron examinadas nuevamente con el fin de llegar a un acuerdo sobre la mejor clasificación del comentario. Durante las revisiones, fueron escasas las diferencias entre asignaciones de un revisador con las del otro.

4.1 Revisión de los datos preliminares

Los datos preliminares consisten de 1374 comentarios javadoc formando un total de 8876 frases. Cada uno de estos comentarios consiste de un bloque de comentario multilínea correspondiente a un método, campo o clase como en el ejemplo a continuación:

```
/**
 * Constructs a new empty <code>ArrayStack</code>. The initial size
 * is controlled by <code>ArrayList</code> and is currently 10.
 */
```

Los datos preliminares son extraídos de la documentación oficial de Eclipse JFace y Apache Commons abarcando 151 clases de JFace y 17 clases de Apache Commons.[10]

[10] Lista con todas las clases abarcadas puede ser encontrada en https://github.com/gabocorrea/Datos-Publicos-del-Trabajo-de-Titulo/

La intención inicial de revisar estos datos era obtener frases de clase no-directivas y unirlas con las directivas del estudio de Monperrus para obtener un set de datos balanceado – con el mismo número de directivas que de no-directivas – con el cual realizar aprendizaje de máquina y Text Mining. Este plan consistía en revisar 2000 comentarios de cada una de las tres API: JFace, Apache Commons y Java. Cada comentario puede tener varias frases y cada frase era asignada a una categoría durante la revisión. El tiempo no fue suficiente para terminar la revisión de los 6000 comentarios y finalmente se obtuvo un conjunto de 1374 comentarios revisados, abarcando a JFace, poco de Apache y nada de la librería Java.

Por ser un campo nuevo la aplicación de Text Mining a comentarios de API, el proyecto no tenía garantías de obtener resultados interesantes usando Machine Learning por lo que se usaron los datos preliminares para realizar una serie de experimentos de Text Mining en Weka para tantear los resultados. Los resultados se encuentran en la sección 6.1

4.1.1 Método de Revisión de Datos Preliminares

Como fue mencionado anteriormente, la asignación manual de comentarios a directivas y no-directivas, no tiene la cualidad de ser un proceso objetivo. Es por esta razón que la asignación de comentarios fue realizada por dos personas para respaldar la validez de las elecciones. Los datos preliminares fueron revisados por el autor del trabajo y por el profesor guía asociado a apoyar el trabajo, Dr. Romain Robbes[11] profesor en el Departamento de Ciencias de la Computación de la Universidad de Chile.

La metodología usada para realizar la revisión de comentarios de los datos preliminares es como sigue. A cada frase, perteneciente a un comentario de API, se le asigna una de las siguientes clases:

- *No-directiva* si es un comentario normal; no es una instrucción crucial o importante
- *Directiva* si es un comentario importante; una instrucción sobre el correcto uso de la API
- *Por-revisar* si es un comentario ambiguo que será revisado posteriormente por el grupo de revisadores

Luego de recorrer todas las frases, se procede a comparar los resultados. De este modo, para cada frase que revisan hacen lo siguiente:

- Si la frase fue asignada con la clase por-revisar, ya sea por uno o ambos de los revisadores, se procede a estudiar con mayor detalle la frase y finalmente se le asigna la clase no-directiva o la clase directiva.
- De otro modo, si sólo uno escogió la clase no-directiva y el otro la directiva, los revisadores conversan para llegar a un acuerdo, y eligen una de esas clases.
- De otro modo, si ambos escogieron la clase no-directiva o la clase directiva, existe un acuerdo y se deja esa clase asignada a la frase.

4.1.2 Observaciones sobre los datos preliminares

Como ya fue indicado, los datos preliminares fueron usados en experimentos de Text Mining para conseguir información temprana de los resultados. Durante estos experimentos preliminares se observaron algunos problemas relacionados con el conjunto de datos, los cuales son comentados a continuación.

[11] http://users.dcc.uchile.cl/~rrobbes/

1 - No uso de todas las API

En los datos preliminares, no se incluyen datos de la librería Java, y hay gran cantidad de comentarios correspondientes a la API de JFace, pero pocos comentarios de Apache Commons. Los datos preliminares no tienen una distribución justa entre las tres librerías usadas. Como se desea lograr una forma de adivinar directivas sobre cualquier API, usar principalmente los datos de JFace es una forma menos objetiva de realizar los entrenamientos de los clasificadores. Lo deseable es usar las documentaciones de tres librerías ampliamente usadas como base para aprender a detectar directivas de otras API, y se espera que estas no sean muy distintas en vocabulario de las tres librerías usadas. Esta es una de las razones de porque se prefieren los datos finales a los preliminares.

2 – No uso de todas las clases ni paquetes

Los comentarios fueron revisados en orden: primero todas las clases de un paquete, luego las del siguiente paquete. Seguir este orden, junto con el no uso de la totalidad de los datos, significa que sólo se consideraron algunas clases de Apache Commons y JFace (recordando que Java no fue incluido en los datos). Esto empeora aún más la distribución de los comentarios. Por esta razón, los datos preliminares no son una correcta representación de los comentarios de API.

3 – Frases repetidas y overfitting

Un inconveniente de revisar los comentarios en el orden mencionado, es que se encuentran muchas frases similares y a veces idénticas. Comentarios de la misma clase o paquete suelen contener comentarios parecidos. Incluso hay ciertos comentarios de API que se repiten globalmente en toda una librería. La revisión de comentarios en orden puede causar *overfitting* en los modelos de predicción de Machine Learning. En Data Mining, overfitting puede ocurrir si los datos usados de entrada son muy específicos a cierto dominio y funcionan muy bien para predecir correctamente ese mismo tipo de datos, pero no obtienen buenos resultados al predecir datos más generales, los cuales suelen distribuirse por todos los dominios posibles del asunto siendo analizado. En este caso, si solamente se usan algunas clases de JFace para entrenar a los clasificadores, se puede obtener un buen rendimiento detectando las directivas de otras clases similares de JFace, pero se obtiene un rendimiento pobre sobre Java, Apache Commons u otras librerías. Siempre se aspira a evitar la presencia de overfitting en Data Mining. Para solucionar este problema existente con los datos preliminares, en los datos finales se trabaja con un subconjunto aleatorio de cada API para reducir la posibilidad de que ocurra overfitting.

4 – Errores al separar comentarios en frases

Por otra parte, los datos preliminares tienen algunas frases separadas de forma incorrecta. Hay varias frases que están incorrectamente separadas en dos, tres o más partes. Por ejemplo, al separar las frases, correctamente se decidió comenzar una nueva frase al encontrar @param o @return, pero incorrectamente se hace lo mismo al encontrar @link. Un ejemplo es el siguiente comentario Javadoc que debía ser separado en las 4 frases que lo componen.

```
/**
 * Makes the given shell resizable on all platforms. The shell must use a {@link GridLayout}.
 * If the shell is not resizable, this method enlarges the {@link GridLayout#marginWidth
 * marginWidth} and {@link GridLayout#marginHeight marginHeight} and expects that the area
 * is not being shrunken or used in any way by other parties.
 *
 * @param shell the shell
 */
```

Pero el comentario anterior fue incorrectamente separado en las siguientes 8 frases:

> Makes the given shell resizable on all platforms.
> The shell must have a {
> @link GridLayout}.
> If the shell is not resizable, this method enlarges the {
> @link GridLayout#marginWidth marginWidth} and {
> @link GridLayout#marginHeight marginHeight} and expects that the area is not being
> shrunken or used in any way by other parties.
> @param shell the shell

Las frases incorrectamente separadas ensucian los resultados en la aplicación de Text Mining. Además son tediosas de clasificar manualmente en CHi y no representan a frases reales de una documentación normal.

4.2 Revisión de los datos finales

Los datos finales consisten de 1205 comentarios javadoc, conteniendo 5903 frases, extraídos de las documentaciones de API de JFace, Apache Commons y Java. A diferencia de los datos preliminares, esto datos incluyen las tres API tomando alrededor de 400 comentarios de cada librería, extraídos de forma aleatoria.

En los datos finales no se incluyen los comentarios de paquetes, clases y campos; sólo se consideran los comentarios de métodos. Se notó que los comentarios de campos suelen ser *no-directivas* formadas por solo una o dos frases, y que los comentarios de clases o paquetes suelen ser complicados y bastante extensos. Además, clasificar frases de comentarios de clase o paquete puede ser complicado, porque en algunas ocasiones es necesario comprender profundamente la clase y la relación que tiene con el resto de la librería. Los comentarios de métodos parecen ser los más abundantes y enriquecedores para el aprendizaje de máquina, al menos en la documentación de las tres librerías consideradas. Además los comentarios de métodos son los más relevantes para una herramienta similar a eMoose (ver 2.1.2), la cual probó ser útil y podría beneficiarse de la detección semi-automática de directivas.

Además, las frases que comienzan con el javadoc tag *@throws* o *@exception* se encuentran omitidas de los datos finales. Es decir, un comentario con *@throws* en la última frase, por ejemplo, es dejado con todas sus frases menos la última. Este tipo de frases suelen ser suficientemente importantes como para ser destacadas, pero se decidió clasificarlas como *no-directivas* pues ya tienen un javadoc tag que las identifica. Un lector de la documentación sabe que debe revisar las líneas que contienen *@throws* o *@exception* para así informarse de las excepciones que podría ocasionar un método. No están incluidas en los datos finales para disminuir el ruido de datos no constructivos en el proceso de Text Mining.

Además, todos los textos Javadoc con dos o menos frases fueron omitidos. Estos comentarios suelen pertenecer a métodos muy simples que realizan una acción pequeña y suelen ser no-directivas. Muchos de ellos suelen ser *setters* o *getters*: funciones que se encargan de modificar o retornar un valor, respectivamente.

4.2.1 Método de revisión de los datos finales

A diferencia de los datos preliminares, y debido al alto acuerdo previo, estos datos no fueron manualmente revisados por ambos revisadores, sino que se realizó una repartición de las API. La documentación de Java fue revisada por Romain Robbes. La documentación de Apache Commons y JFace fue revisada por el autor del trabajo.

En estos datos las frases son asignadas a una de las clases *no-directiva*, *directiva*, *semi-directiva* o *directiva-null*. Una frase es asignada a *semi-directiva* si es similar a un consejo o a una instrucción que no necesariamente es directiva. Esta clase de comentarios de API corresponde a un segundo nivel de importancia de instrucciones, después del primer nivel de importancia que son los comentarios de clase directiva. Un ejemplo de *semi-directiva* es: *"This method is useful for implementing <code>List</code> when you cannot extend AbstractList"*. Por otra parte, las frases asignadas a *directiva-null* son comentarios que tratan sobre el uso del valor especial *null*. Estos comentarios suelen advertir si algún parámetro de la función puede ser null o no, o que cierto método podría retornar el valor null. Para determinar si una frase es *directiva-null*, la regla seguida fue: si es instrucción importante (o directiva) pero trata principalmente sobre el valor *null* es *directiva-null*. Si es instrucción importante que menciona *null* pero trata principalmente sobre otra advertencia es *directiva*. En otro caso es *no-directiva* (una *semi-directiva* que habla sobre el uso de *null*, se anota como una *directiva-null*). Un ejemplo de *directiva-null* es la siguiente frase: *"@param list the list to generate the hashCode for, may be null"*.

5 Text Mining sobre comentarios de API

El objetivo de realizar Text Mining y usar algoritmos de Machine Learning es lograr obtener un programa que destaque automáticamente las directivas de los comentarios normales de documentaciones de API. Una forma de lograr estas clasificaciones es usar Weka, un ambiente de herramientas para realizar Data Mining y aplicar algoritmos de Machine Learning. En la sección 5.1 se encuentra una breve introducción a Weka.

Sin duda, un programa que clasifique directivas no puede evitar equivocarse en algunos casos porque los comentarios de código fuente en lenguaje natural pueden adoptar millones de combinaciones de palabras. No hay una regla existente que nos pueda determinar con certeza si un comentario es una directiva o si no lo es. Por esto es que solo se pueden realizar aproximaciones, o Machine Learning en nuestro caso, para acercarse a un buen resultado.

En esta sección se describe el diseño y detalle de los experimentos realizados en Weka usando los datos preliminares y datos finales discutidos en la sección 4, los cuales abarcan ejemplos de las API de JFace, Apache Commons y Java. La mayoría de los experimentos buscan encontrar el mejor rendimiento clasificando las directivas, mientras que otros experimentos buscan obtener información intrínseca al dominio de los datos y el problema estudiado.

5.1 Weka

Weka es una plataforma de software para el uso de Machine Learning y Data Mining. Posee una interfaz gráfica como también un API en Java para usar sus funcionalidades. Los componentes de Weka usados en este estudio son su explorador, su experimentador y su API. La Imagen 5.1 e Imagen 5.2 muestran la interfaz gráfica del Explorador y el Experimentador de Weka respectivamente.

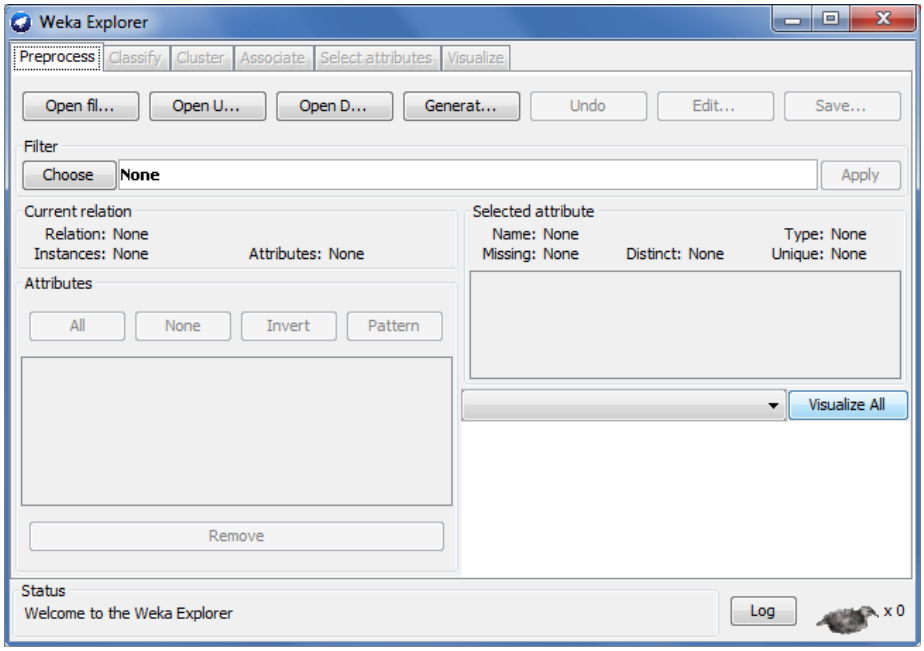


Imagen 5.1: Interfaz gráfica del explorado de Weka.

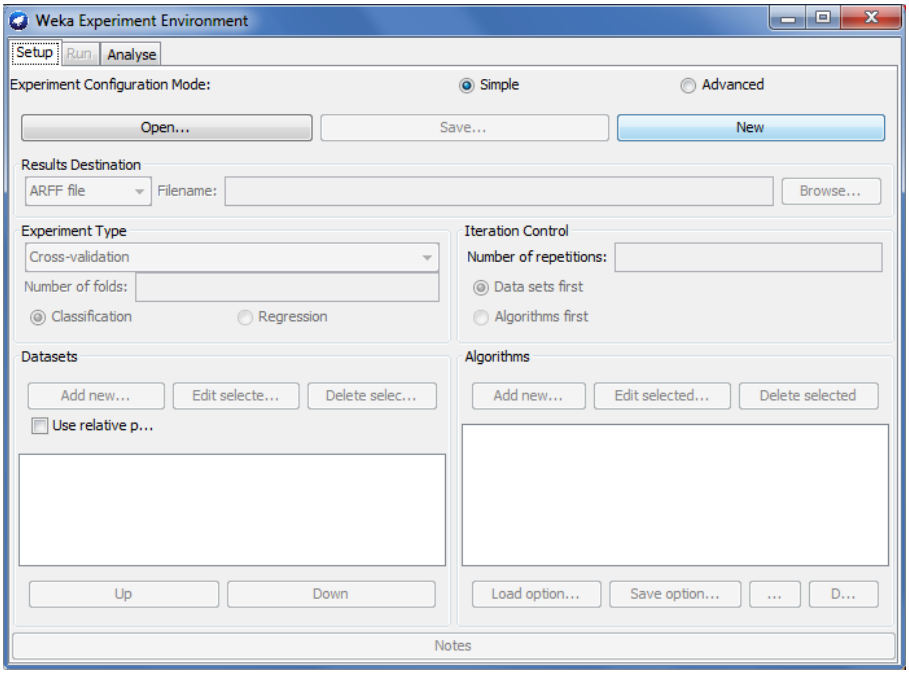


Imagen 5.2: Interfaz gráfica del experimentador de Weka.

El programa es entrenado con datos que deben ser guardados en un archivo *.arff* el cual posee un formato particular, pero muy similar a los archivos *.csv*. Es un archivo con instancias de datos, cada instancia en una línea con valores separados por coma representando un vector de valores[12]. En este trabajo cada instancia es un vector $(string, tipo\_de\_comentario)$ donde $string$ es texto y $tipo\_de\_comentario$ es uno de los valores {*non-directive,directive,semi-directive,null-directive*} o posiblemente solo {*non-directive,directive*}.

5.1.1 Filtros

Weka posee una serie de filtros que facilitan la manipulación de las instancias y atributos de los datos. Es común aplicar un filtro de Weka llamado *StringToWordVector* cuando se trabaja con frases de texto y de hecho es usado en todos los experimentos de este trabajo.

Lo que hace el filtro es transformar cada instancia de texto a una lista de las palabras que forman ese texto. Esto facilita el análisis del texto y de hecho es necesario para entrenar los algoritmos de clasificación. La Imagen 5.3 muestra dos instancias de texto antes y después de haber sido pasado por este filtro. El filtro tiene parámetros que cambian su comportamiento y en este trabajo se ha experimentado con algunos de ellos. Los parámetros usados son descritos en la Tabla 5.1.

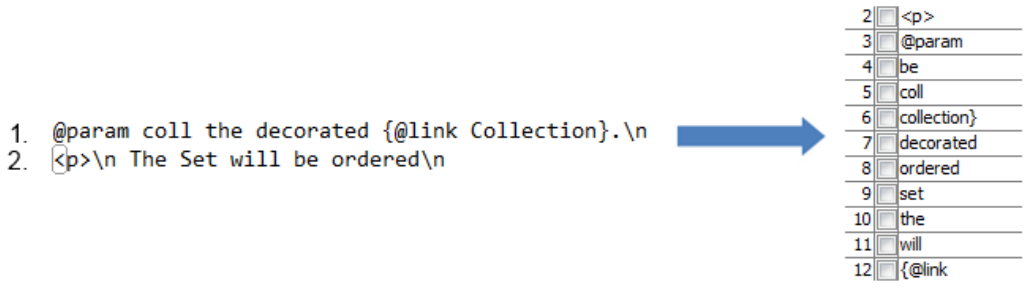


Imagen 5.3: Filtro StringToWordVector siendo aplicado sobre dos instancias de texto, resultando en una lista de las palabras que lo forman.

[12] Referencia oficial para los archivos .arff : http://weka.wikispaces.com/ARFF+%28stable+version%29

Tabla 5.1: Parámetros del filtro StringToWordVector usados en los experimentos realizados

Parámetro	Descripción
attributeIndices	Rango de atributos donde va a ser aplicado el filtro. Siempre toma el valor de '*first'* en los experimentos realizados
lowerCaseTokens	Si es *true* cada palabra resultante queda completamente en minúsculas. *False* no hace ni modifica nada
minTermFreq	Mínima cantidad de veces que debe aparecer una palabra en todas las instancias de los datos filtrados para ser considerada en el vector resultante
outputWordCounts	Si es *true* los vectores tienen valores equivalentes a la cantidad de veces que apareció cada palabra. Si es *false* el vector contiene sólo los valores 0 y 1
stemmer	El algoritmo de *stemming* a usar por el filtro
stopwords	Archivo de texto con lista de palabras *stopwords*
tokenizer	Tokenizer encargado de separar la frase en sus palabras. Puede ser elegido un '*n-gram tokenizer*' que permite separar la frase en grupos de *n* palabras contiguas.
useStoplist	Si es *True,* se aplica el filtro de stop-words usando las palabras apuntadas por el parámetro *stopwords*. Si es *False,* se omite el uso de stop-words.
wordsToKeep	Tamaño aproximado del vector resultante

La mejor forma de aplicar el filtro StringToWordVector es dentro del clasificador '*FilteredClassifier*', el cual aplica el filtro y luego entrena un clasificador especificado por el usuario sobre los datos filtrados.

En la pestaña 'Classify' de la ventana se entrenan los algoritmos de clasificación con los datos pasados al programa. Aquí se elige algún clasificador el cual es entrenado con los datos, y luego del entrenamiento se prueban con un archivo con datos de prueba llamado *test* set o realizando *cross-validation*. Alternativamente se puede entrenar al clasificador con un porcentaje de los datos de entrada y probar su rendimiento con el resto de los datos. Los resultados aparecen en la misma pestaña una vez terminado el análisis. Estos son valores que representan el rendimiento del clasificador entrenado sobre los datos de prueba y son explicados en la sección 5.2.

5.2 Clasificadores y parámetros probados

Los clasificadores y distintos parámetros probados, con el fin de encontrar las combinaciones con mejores rendimientos predictivos, se encuentran en las tablas a continuación.

Clasificadores	Parámetros StringToWordVector	Parámetros de SMO
ZeroR	Stemming	kernel type
OneR	StopWords	c coefficient
PART		
NaiveBayes		
NaiveBayesMultinomial		
BayesianLogisticRegresion		
AdaBoostM1		
DMNBtext		
IBk *(sólo datos finales)*		
Logistic *(sólo datos finales)*		
LibSVM		
SMO		
RandomForest		
J48		

5.3 Evaluación del rendimiento de un clasificador

Cada experimento realizado en Weka entregó resultados los cuales fueron comparados entre sí, para identificar cuál era mejor detectando directivas de API. Pero los resultados entregan varios números de los cuales no todos son igualmente útiles para describir cuán bueno es un clasificador para identificar directivas. Para comparar el rendimiento de los clasificadores se decide considerar un subconjunto de las métricas de los resultados. A continuación se muestra un ejemplo de resultado en Weka (en la Imagen 5.4) y luego se detallan las métricas usadas en este trabajo para evaluar la calidad de los modelos de predicción.

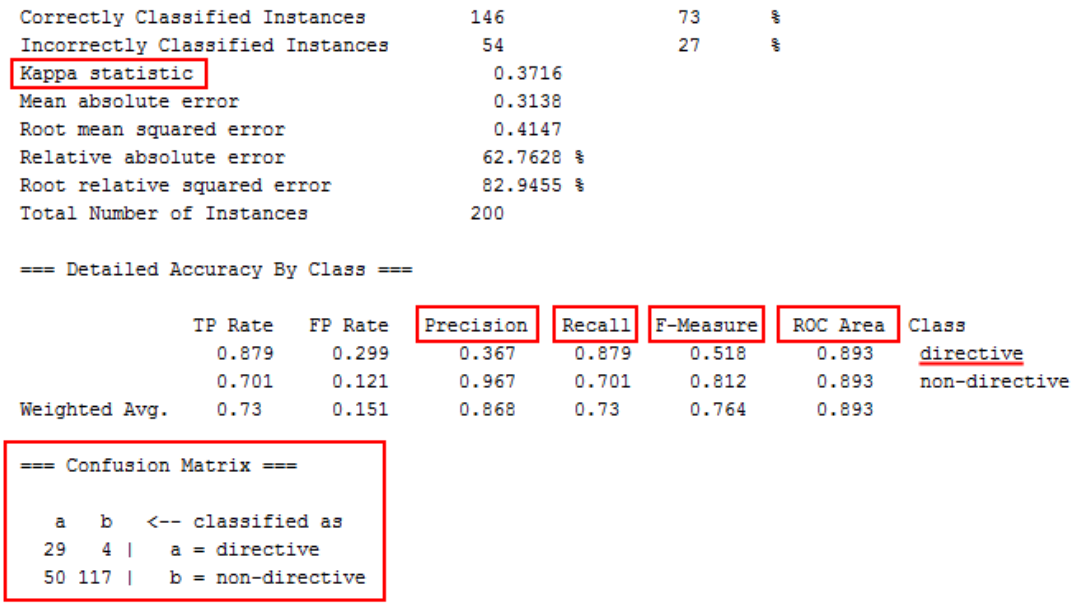


Imagen 5.4: Métricas usadas para evaluar el rendimiento de los clasificadores en este trabajo aparecen destacadas dentro de cuadros en estos resultados de Weka. El enfoque está en medir el *precision, recall y F-Measure* de la clase *directive*.

5.3.1 Métricas principales

En este trabajo se busca formar un modelo que detecte correctamente la mayor cantidad de directivas, manteniendo una baja cantidad de errores. Esto se traduce principalmente en maximizar los valores *Recall* y *Precision* de la clase *directive*. El valor de *precision* de la clase *directive* está relacionado con la clase *non-directive* (como veremos más adelante) y ya entrega la información necesaria para este estudio. En la Imagen 5.4, los valores mínimos necesarios para entender el rendimiento de cierto algoritmo de clasificación de Weka son los de la clase *directive* 0.367 (*precision*) y 0.879 (recall). Puede ser útil ver el valor *F-Measure* de *directive*, el cual relaciona a *precision* y a *recall*.

A continuación se listan las métricas consideradas para evaluar el rendimiento de un clasificador según este trabajo.

Confusion Matrix:

Es una matriz que resume los resultados mostrando cuántas instancias fueron clasificadas por cada clase y a qué clase pertenecían realmente. Las posiciones de la matriz representan lo siguiente:

$$\begin{vmatrix} VP & FN \\ FP & VN \end{vmatrix}$$

VP: Verdadero positivo **FN**: Falso negativo
FP: Falso positivo **VN**: Verdadero negativo

$$\boldsymbol{VP}: directivas\ correctamente\ adivinadas$$

$$\boldsymbol{VN}: no\ directivas\ correctamente\ adivinadas$$

$$\boldsymbol{FN}: directivas\ incorrectamente\ clasificadas\ como\ no\ directivas$$

$$\boldsymbol{FP}: no\ directivas\ incorrectamente\ clasificadas\ como\ directivas$$

y como ejemplo, en la Imagen 5.4 se observa que en los datos de prueba hay 29+4=33 directivas de las cuales 29 fueron clasificadas correctamente y 4 fueron clasificadas

incorrectamente mientras que 50+117=167 son no-directivas de las cuales 117 fueron clasificadas correctamente y 50 fueron clasificadas incorrectamente.

Recall (de la clase 'directive'):

Proporción de instancias correctamente clasificadas como directiva, dividido por el total de instancias que realmente son directivas. Según la definición de la matriz de confusión más arriba es:

$$recall = \frac{VP}{VP + FN}$$

Un clasificador con valor de recall 0.5 en la clase directiva debería identificar aproximadamente la mitad de las directivas. Entonces un valor de recall bajo debe entenderse como un modelo que no se da cuenta de la existencia de varias directivas. De este modo, podría pasar en un caso con recall bajo que varías instrucciones críticas no sean destacadas. En este trabajo es un error muy indeseado porque durante la post-revisión manual en CHi un usuario puede corregir las frases destacadas incorrectamente (i.e. que deben ser de tipo *no-directiva*) pero no puede destacar las frases que no fueron correctamente clasificadas como directiva (secciones 3.1.1 y 3.2.1)

Precision (de la clase 'directive'):

Proporción de instancias que realmente son de clase A dividido por el total de instancias clasificadas como A. Según la definición de la matriz de confusión más arriba es:

$$precision = \frac{VP}{VP + FP}$$

Un clasificador con valor de *precisión* 0.5 en la clase *directiva* resultaría en que de todas las frases de API destacadas, aproximadamente la mitad sería *no-directivas* y la otra mitad serían realmente *directivas*. Es decir, muchas frases destacadas no serán realmente directivas. Este tipo de error es indeseado pero menos que los errores asociados a un *recall* bajo, porque durante la post-revisión manual en CHi un usuario si podrá corregir estos errores al revisar las frases destacadas. Es un error menos grave, pero si el error es muy grande la corrección manual puede tardar mucho tiempo, por lo que sí es importante que las precisiones no sean muy bajas.

F-Measure de la clase 'directive':

Un valor que representa promedio ponderado de los valores *recall* y *precision.*

$$F\ Measure = \frac{2 * recall * precision}{recall + precision}$$

5.3.2 Métricas secundarias

Además de las métricas primarias es importante revisar los valores ROC Area y Kappa Statistic pues entregan información estadísticamente corregida de los resultados. ROC Area es una métrica común en Data Mining mientras que Kappa Statistic es una métrica común en estadística. Lo que se busca es que estos valores no resulten ser muy menos que 1, pues esto indicaría algún posible problema con la validez estadística de los datos.

ROC Area:

Los mejores clasificadores se aproximan al valor 1, mientras que un valor de 0.5 es comparable a adivinar las clases por azar.

Es una medida de la exactitud de un clasificador. Se calcula tomando el área bajo la curva del gráfico de la recall (eje y) vs la proporción de FP (eje x) con $proporción\ de\ FP = \frac{FP}{(FP+VN)}$.

Kappa statistic:

Un valor mayor a 0 significa que el clasificador es mejor que adivinar al azar, y su valor máximo es 1.

5.4 Criterio de comparación entre modelos de clasificación en Weka

Lo principal es maximizar lo más posible el valor de recall. Después de eso, se desea que el valor de precision sea alto y siempre mayor a 0.5. Además se requiere que ROC Area y Kappa Statistic sean en lo posible mayor a 0.75 y 0.5 respectivamente. El valor de F-Measure, el cual depende solo de recall y precision, suele ser mayor a 0.7 cuando un resultado es bueno. Alternativamente se puede maximizar recall y después maximizar F-Measure, lo cual tiene un efecto similar a medir recall y precision.

Hay una excepción a la regala de maximizar recall. Esto puede ocurrir si un resultado presenta un valor muy alto de F-Measure y al clasificador se le puede aplicar una matriz de costos. Esto es tratado en mayor detalle en las secciones 6.2.1 y 6.2.6.

Un lector, al no comprender en detalle el significado de recall, F-Measure, ROC Area puede necesitar alguna referencia que le sirva de guía para saber si un resultado es bueno, regular o malo. La Tabla 5.2 muestra un ejemplo de cuáles resultados son mejores que otros para dar a conocer los resultados que satisfacen de mejor manera a los objetivos de este trabajo, que es detectar la mayor cantidad de directivas manteniendo bajo control el número de falsos positivos.

Tabla 5.2: Ejemplos de cuáles resultados serían los mejores según los objetivos de este trabajo.

Recall	*F-Measure*	*Precision*	Calidad del resultado
0.97	0.74	0.6	excelente
0.93	0.77	0.66	excelente
0.89	0.78	0.7	bueno
0.89	0.75	0.65	bueno
0.8	0.80	0.8	bueno/regular
0.6	0.55	0.5	malo
0.4	0.48	0.6	muy malo

5.5 *Métricas no usadas:*

Otros valores no fueron usados porque no representan fielmente a la calidad de un clasificador para detectar directivas. La Imagen 5.5 más adelante muestra los resultados de un clasificador que puede parecer muy bueno a simple vista pero no necesariamente lo es. El valor '*Correctly Classified Instances*' es muy bueno solo porque los datos de prueba tienen muchas más frases normales que directivas, pues es un set de datos *desbalanceado*, y el clasificador identificó muchas frases normales pero pocas directivas. De hecho es fácil identificar las no-directivas, basta con usar Zero-R, un clasificador que solo elige la clase más común para cada frase resultando en un 83.5% de frases correctamente clasificadas. Mirando la matriz de confusión vemos que 167 no-directivas fueron correctamente clasificadas, 0 no-directivas fueron incorrectamente clasificadas como directivas, 21 directivas fueron correctamente clasificadas y 12 de ellas fueron incorrectamente clasificadas como no-directivas. Es un modelo que no detecta muy bien las directivas: *recall* de directive tiene un valor bajo de 0.636 pero por otra parte tiene una buena precisión de 1. Si solo miráramos ROC Area y *correctly classified instances* no se notaría que el modelo no detecta bien a las directivas. Además, si se usaran los valores de la fila '*Weighted Avg.*' no se notaría que hay un *recall* de solo 0.636 para las directivas. Por estas razones se usa el subconjunto de métricas detallado antes.

```
Correctly Classified Instances         188               94      %
Incorrectly Classified Instances        12                6      %
Kappa statistic                          0.7451
Mean absolute error                      0.06
Root mean squared error                  0.2449
Relative absolute error                 20.3109 %
Root relative squared error             65.7825 %
Total Number of Instances              200

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 0.636     0          1         0.636     0.778      0.818    directive
                 1         0.364      0.933     1         0.965      0.818    non-directive
Weighted Avg.    0.94      0.304      0.944     0.94      0.934      0.818

=== Confusion Matrix ===

  a   b   <-- classified as
 21  12 |   a = directive
  0 167 |   b = non-directive
```

Imagen 5.5: Resultados de un clasificador sobre los datos preliminares (desbalanceados) para servir de ejemplo de que algunas métricas que no representan la calidad, según los objetivos de este trabajo, de un clasificador para detectar directivas.

6 Resultados

En esta sección se detallan los resultados de los experimentos realizados en Weka, que tienen el propósito de encontrar un buen modelo para adivinar las directivas dentro de una documentación de API.

Para comparar la efectividad de cada modelo se usa la metodología explicada en la sección 5.2. Todos los experimentos fueron realizados en Weka 3.6.12 en Windows 7 con 3 GB de memoria RAM y procesador i5 M430 2.27 GHz. Se usó Java versión 1.8.0_51 configurado con una memoria máxima de 989.9MB en tiempo de ejecución.

6.1 Resultados Preliminares

Los resultados preliminares provienen de experimentos realizados en el Explorador de Weka usando los datos preliminares descritos en la sección 4.1. Los experimentos preliminares fueron realizados con la siguiente configuración:

Tabla 6.1: Configuración usada para obtener los resultados preliminares.

Datos de prueba	200 frases extraídas al azar del set de datos preliminar
Datos de entrenamiento	8876 - 200 = 8676 frases del set de datos preliminar
Clasificador	FilteredClassifier con filtro StringToWordVector (ver sección 5.1.1) y con todos los siguientes clasificadores: ZeroR,OneR,PART,NaiveBayes,NaiveBayesMultinomial, BayesianLogisticRegresion,AdaBoostM1,DMNBtext, LibSVM,SMO,RandomForest,J48.
StringToWordVector	Usando los parámetros por defecto de Weka, excepto por: • *attributeIndices=first* • *lowerCaseTokens=true*
Clases asignadas a los comentarios	directiva, no-directiva (durante la revisión manual, algunos comentarios fueron marcados como *por-revisar* , pero estos fueron revisados nuevamente, quedando como *directiva* o como *no-directiva*)

La Tabla 6.2 muestra los rendimientos de los clasificadores sobre los datos preliminares. Los valores *F-Measure*, *recall*, *precision* y *ROC-Area* corresponden a la clase *directiva*, conforme a los valores relevantes de los resultados descritos en la sección 5.3.1. El clasificador *ZeroR* asigna la clase *no-directiva* a todas las frases de comentarios, pues hay más frases de tipo *no-directiva* que de tipo *directiva* en las documentaciones de API estudiadas por este trabajo. De este modo, *ZeroR* consigue el peor rendimiento posible, y es incluido pues es costumbre tenerlo como base de comparaciones durante actividades de *data mining*.

Tabla 6.2: Rendimiento de clasificadores sobre los datos preliminares. Los mejores valores de algunas columnas están resaltados.

Clasificador	F-Measure	Recall	Precision	ROC Area	Kappa Statistic
ZeroR	0	0	0	0.50	0
AdaBoostM1	0.54	0.52	0.57	0.88	0.45
DMNBtext	0.63	0.52	**0.81**	0.81	0.58
NaiveBayesMultinomial	0.65	0.64	0.66	0.87	0.58
OneR	0.78	0.64	**1**	0.82	0.75
BayesianLogisticRegresion	0.70	0.67	0.73	0.81	0.06
RandomForest	0.76	0.67	**0.88**	0.97	0.72
PART	0.76	0.76	0.76	0.89	0.71
NaiveBayes	0.64	**0.82**	0.53	0.90	0.55
LibSVM	**0.81**	**0.82**	**0.79**	0.89	0.77
SMO	**0.81**	**0.82**	**0.79**	0.89	0.77
J48	**0.81**	**0.85**	**0.78**	0.96	0.77

Estos resultados indican que se obtuvieron mejores resultados con LibSVM, SMO y J48. NaiveBayes tiene un muy buen *recall*, pero tiene bajo *precision*. OneR tiene bajo *recall*, pero un *precision* igual a 1, es decir, sin ocurrencias de falsos positivos.

Los resultados de los mejores clasificadores son buenos porque están adivinando hasta un 81% (*recall*) de las directivas y, de todas las clasificadas como directivas, se obtiene hasta un 79% (*precision)* que son realmente directivas y solo un 21% que fueron incorrectamente tomadas como directivas.

Además, se realizaron algunas pruebas extras con *NaiveBayes* y *SMO*, variando la lista de stop-words y algunos parámetros de *SMO*. Sin embargo no se obtuvieron resultados significativamente mejores. Estos resultados se incluyen en el Anexo sección 9.1. Ocurrió lo mismo con otras pruebas comparando entre los stemmers *Lovins*, *Iterated Lovins* y *Snowball*. De este modo, no se encontró evidencia en estas pruebas de que usar *stop-words*, *stemmers* o variar parámetros de *SMO*, pueda mejorar el rendimiento de *NaiveBayes* o *SMO*.

Los resultados con los datos preliminares sirvieron para obtener una mayor comprensión del rendimiento de algoritmos de *Text Mining* sobre comentarios de API. Pero al mismo tiempo, revelaron una serie de problemas con los datos preliminares, los cuales fueron descritos en la sección 4.1.2. Estos resultados son obtenidos usando sólo 200 instancias de prueba y en contraste, entrenando con un mucho mayor número de 8676 instancias. Además, no dan información sobre si es factible adivinar directivas de otras librerías API no incluidas en el set de entrenamiento, ni tampoco del porcentaje de API requerido para entrenar a un clasificador y obtener resultados satisfactorios. Por estas razones fue necesario realizar experimentos con los datos finales. Los resultados con los datos finales se encuentran a continuación.

6.2 Resultados Finales

A continuación se presentan varias combinaciones entre sets de entrenamientos y de pruebas usados para realizar los experimentos. El objetivo principal de la mayoría de los experimento es encontrar los mejores clasificadores para adivinar correctamente la mayor cantidad de *directivas* maximizando *recall* y *F-Measure*. El otro objetivo, es descubrir el porcentaje de API necesario durante el entrenamiento de los clasificadores, para obtener resultados aceptables.

Los experimentos usaron los datos preliminares descritos en la sección 4.2 y fueron realizados con la siguiente configuración:

Datos de prueba	Variado: dos API, una API, porcentaje de una API, etc.
Datos de entrenamiento	Apache: 1623 frases sacadas de 380 comentarios Java: 2370 frases sacadas de 416 comentarios JFace: 1910 frases sacadas de 409 comentarios
Clasificador	FilteredClassifier con filtro StringToWordVector (ver sección 5.1.1) y con todos los siguientes clasificadores: ZeroR,OneR,PART,NaiveBayes,NaiveBayesMultinomial, BayesianLogisticRegresion,AdaBoostM1,DMNBtext,IBk, Logistic,LibSVM,SMO,RandomForest,J48.
StringToWordVector	Parámetros por defecto de Weka, excepto por: • *attributeIndices=first* • *lowerCaseTokens=true* Estos parámetros fueron constantes a lo largo de todos los experimentos. Otros parámetros fueron, sin embargo, variados con el fin de realizar experimentación con variadas combinaciones.
Clases asignadas a los comentarios	directiva, no-directiva (durante la revisión manual, algunos comentarios fueron marcados como *semi-directiva* y otros como *directiva-null*, pero estos fueron finalmente considerados como partes de la clase *directiva*)

6.2.1 F-Measure vs Recall

Hay casos donde maximizar *F-Measure* puede resultar mejor que maximizar *recall*. Incluso cuando el objetivo es acercar el valor de *recall* a 1, puede ocurrir que usando lo que es llamado una *matriz de costos* se logren mejores resultados. De los experimentos realizados se obtuvo información que apoya a la afirmación anterior.

En este estudio se quiere buscar un clasificador que tenga la menor cantidad de *falsos negativos*, o un *recall* muy cercano a 1 como se explica en las secciones 3.1.1 y en 3.2.1, significando que hay que optimizar el valor de *recall*. Pero los resultados de un experimento, mostrados en la

Tabla **6.3**, muestran que optimizar *F-Measure* permite lograr resultados con menos *falsos negativos* si posteriormente se usa la *matriz de costos*. En la tabla, *NaiveBayes* obtiene 0.93 de *recall*, superando a *SMO* y *RandomForest* pero por otro lado tiene un valor más bajo de *precision* y *F-Measure que los otros clasificadores*. Al usar matrices de costo con costos de 15 y 4 respectivamente para *SMO* y *RandomForest*, se obtienen en ambos un recall de 0.94, superando a *NaiveBayes* (se usan los costos de 4 y 15 pues con estos se logró superar a *Naive*Bayes en su valor de *recall*). Al aplicar la matriz de costos sobre estos clasificadores, sus valores de *precision* bajan, pero de todas formas en este experimento, el *precision* de *NaiveBayes* sigue siendo más bajo que los otros clasificadores. Notemos que los valores de *F-Measure* de SMO y RandomForest son mayores al de NaiveBayes, lo cual les permite comportarse de mejor manera con las matrices de costos por la relación inversa que existe entre *precision* y *recall* al usar la matriz de costos.

La matriz de costos representa a los resultados de la matriz de confusión $\begin{bmatrix} VP & FN \\ FP & VN \end{bmatrix}$. De este modo, las matrices usadas sobre SMO y RandomForest fueron $\begin{bmatrix} 0 & 4 \\ 1 & 0 \end{bmatrix}$ y $\begin{bmatrix} 0 & 15 \\ 1 & 0 \end{bmatrix}$ respectivamente. La matriz de costos intenta desincentivar la clasificación de las instancias de los falsos negativos en el caso del experimento realizado, asignándole un costo mayor a este tipo de errores.

Tabla 6.3: **Resultados con datos finales (Sólo NaiveBayes, SMO y RandomForest). Set de entrenamiento: Apache Commons, Java y el primer 10% de JFace. Set de prueba: 90% restante de JFace.**

	F-Measure	recall	precision	ROC Area	kappa
NaiveBayes	0.54	**0.93**	0.38	0.91	0.43
SMO	0.66	0.87	0.53	0.88	0.59
RandomForest	0.71	0.77	0.67	0.96	0.67
SMO (*matriz con costo de 15 sobre los falsos negativos)*	0.60	**0.94**	0.44	0.88	0.51
RandomForest (*matriz con costo de 4 sobre los falsos negativos*)	0.65	**0.94**	0.50	0.96	0.58

También se observa que al aplicar la matriz de costos sobre un clasificador empeora su valor de *F-Measure*, lo cual se puede notar viendo el cambio en *F-Measure* de *SMO* y *RandomForest* en la Tabla **6.3**.

Un problema con la matriz de costos es que hay clasificadores que no necesariamente mejoran su valor de *recall* e incluso pueden empeorar al usar la matriz de costos, como se estudia en la sección 6.2.6.

6.2.2 Usando Keywords

Este experimento consiste en evaluar las tres librerías de los datos finales usando la regla de los *keywords*: 'Si una frase contiene al menos 1 *keyword* entonces es directiva y si no es así, es una no-directiva'.

Se usan los *keywords* de estudio de Monperrus (set #1) y otro conjunto igual pero conteniendo además el *keyword* '*null'* (set #2), listados a continuación.

Tabla 6.4: Listado de *keywords* que determinan si un comentario es o no es una directiva. El set #1 incluye a todas las palabras menos 'null' y el set #2 considera a todas las palabras.

addition*	call*	fast	note*	quick	strong*
after	concurren*	inherit*	once	recommend*	subclass*
alternativ*	condition*	invo*	only	reimplement*	super*
assum*	debug*	lock*	overload*	require*	synchron*
aware*	desir*	mandat*	overrid*	restrict*	thread*
before	efficien*	may	overwrit*	shall	warn*
best	encourage*	must	performan*	should	***null (set #2)***
better	error*	necessar*	portab*	simultaneous*	
between	extend*	never	prior	strict*	

Se decide incluir el *keyword* 'null' porque durante la preparación manual de los datos usados se marcó la gran mayoría de las frases que contenían esta palabra como *directiva-null*, una subclase entre las directivas. Además, las palabras con * pueden terminar en cualquiera de sus variaciones y también serán consideradas como *keywords*.

Tabla 6.5: Resultados de clasificar comentarios de cada API usando la regla de los *keywords*.

	Set #1 Keywords			Set #2 Keywords (con null)		
	recall	precision	f-measure	recall	precision	f-measure
Apache Commons	0.25	0.65	0.37	0.49	0.69	0.57
Java	0.43	0.64	0.52	0.57	0.69	0.62
Jface	0.58	0.55	0.57	0.91	0.63	0.75
(Promedio)	0.42	0.62	0.48	0.65	0.67	0.65

Los resultados de la Tabla 6.5 muestran que el rendimiento fue mejor con el set #2 de *keywords* lo cual se explica con el hecho de que hay varias frases que fueron marcadas como directivas durante las revisiones manuales.

Un resultado interesante es que el valor de *recall* mejoró más en JFace que en las otras librerías al incluir la palabra *null* en el set de *keywords*. Alcanza un alto valor de 0.91. Esto sin dudas indica que JFace tiene una proporción mayor de directivas que incluyen la palabra *null* que las otras API.

Otra observación es que los resultados en promedio de las tres librerías no es suficientemente bueno según la escala de rendimiento en 5.3 y no supera la calidad de otros clasificadores que usan aprendizaje de máquina como se ve en resultados de más adelante. Sin duda en JFace los

resultados con *keywords* son excelentes, pero no resultó ser así con Apache Commons ni con Java.

Posiblemente la estrategia de keywords puede ser mejorada para obtener mejores resultados. Sin embargo tiene la desventaja que no puede ser usada con la matriz de costos para acercar más el valor de *recall* hacia 1, al menos en su algoritmo básico usado en este trabajo.

6.2.3 Entrenando y probando con la misma API

Este experimento consiste en usar una API y entrenar a los clasificadores con un porcentaje de ella y luego probar con el resto de la API. Esto se realizó para las 3 API consideradas en el trabajo. El orden de las instancias de cada API es aleatorizada antes de realizar la partición entre el set de entrenamiento y el de prueba. El objetivo de este experimento es observar si se puede adivinar las directivas de una librería entrenando con un porcentaje muy bajo de los datos, no teniendo algún modelo previo de cómo detectar las directivas. Buenos resultados en este experimento podría hacer posible clasificar manualmente muy pocas directivas de una documentación y detectar el resto automáticamente.

Los resultados se encuentran graficados en la Imagen 6.1 (el clasificador LibSVM fue omitido de los gráficos por obtener resultados idénticos a SMO). Es de interés encontrar los clasificadores con *recall* cercano a 1, comprobando al mismo tiempo que *F-Measure* no sea muy menor a 1. En la primera columna se encuentran los gráficos correspondientes a *recall* y la otra columna se muestran los gráficos que corresponden a *F-Measure*. Un *recall* alto fue obtenido al entrenar porcentajes de solo 2.5% con el clasificador Logistic en las API de Apache y JFace, y con el clasificador NaiveBayesMultinomial en la API de Java. Por el tamaño de las API, al hablar de 2.5% de ellas significa entrenar con solo 40 o 60 instancias aproximadamente. Por otro lado, Logistic en los gráficos de la primera y tercera fila, termina con peor rendimiento que los otros clasificadores al alcanzar porcentajes de 20% o 30%.

Pero no hubo un clasificador que consistentemente fuera bueno entrenando con un 2.5% o 5% de los datos por lo que no se puede concluir si hay un clasificador útil para el trabajo actual si se entrena con muy pocas instancias. *Logistic* obtuvo un valor de solo 0.4 en *recall* en la API de Java al entrenar con 2.5%, pero obtuvo los mejores rendimientos de *recall* en Apache y JFace. Estos resultados pueden indicar que fue una casualidad que haya ocurrido esto con Logistic y con NaiveBayesMultinomial. Por otra parte, puede ser que realizando una futura investigación más extensiva usando un 2.5% de una API como entrenamiento, se obtengan resultados con clasificadores que obtienen un buen rendimiento de predicción.

En general los resultados no son muy prometedores. Sólo en los datos de JFace se logró obtener un *recall* mayor a 0.85 en 30% y un *F-Measure* cercano a 0.6. Estos resultados no logran apoyar que es suficiente entrenar con un porcentaje menor a 30% de una API para lograr predecir directivas en el porcentaje restante de una API, sin considerar un modelo de predicción previo. Sin embargo, estos resultados no consideran el uso de las matrices de costo, las cuales posiblemente podrían mejorar el rendimiento de *recall*, y trabajo futuro sobre esto se podría realizar.

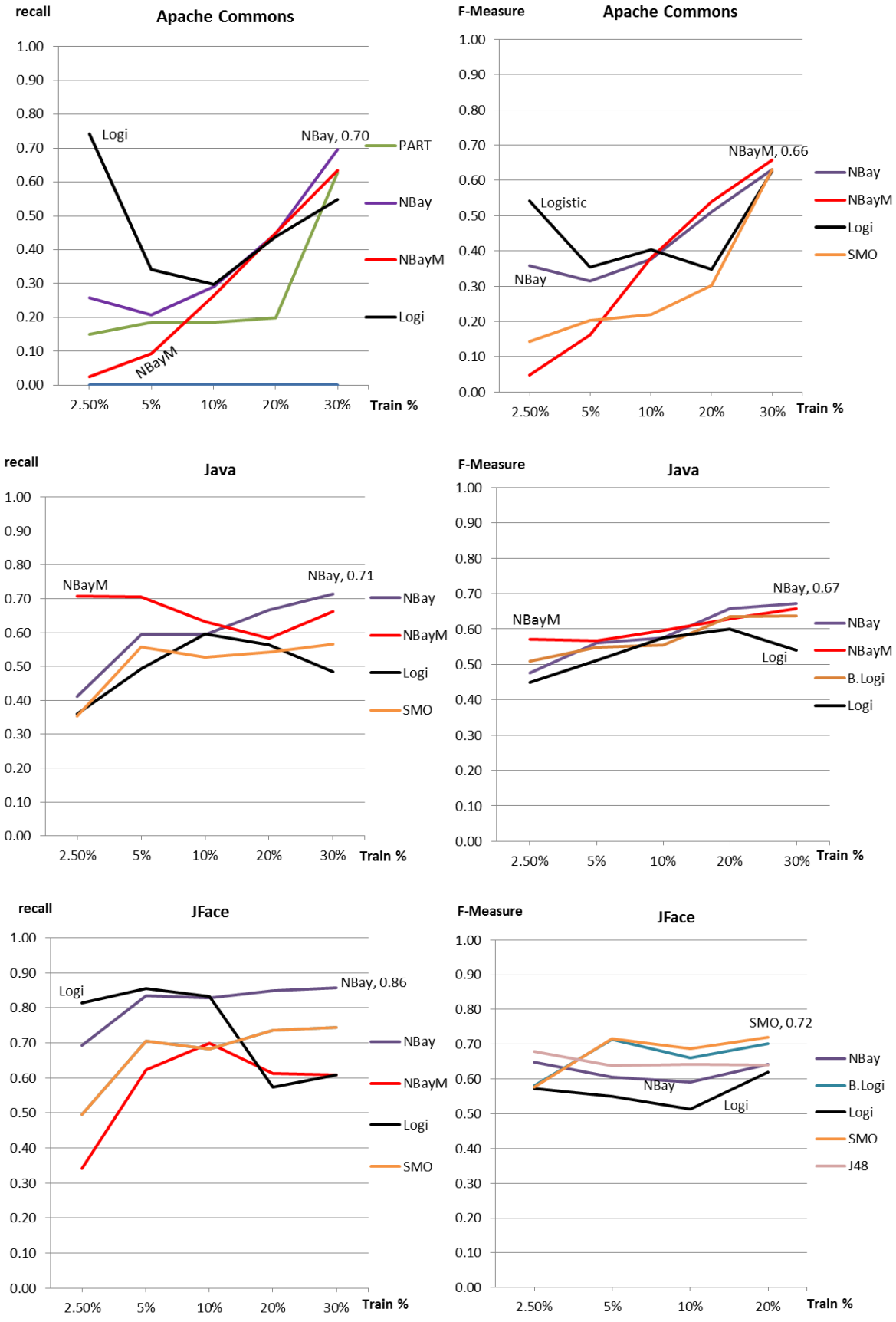


Imagen 6.1: Entrenando con un porcentaje de cada API y usando el resto de los datos como set de pruebas.

6.2.4 Entrenando con 1 API y probando con 2 API

Se realizaron pruebas entrenando con una API y realizando pruebas en las 2 librerías restantes, haciendo esto por cada librería formando 3 resultados. Las siguientes tablas muestran los resultados. Se encuentran ordenadas en orden creciente por *recall* y los mejores valores aparecen destacados.

Tabla 6.6: Resultados de entrenar clasificadores con Apache Commons y probar sobre Java y JFace.

Clasificador	precision	recall	f-measure
ZeroR	0.00	0.00	0.00
RandomForest	0.82	0.06	0.11
IBk	0.64	0.11	0.19
OneR	0.46	0.15	0.22
J48	0.46	0.39	0.42
Logistic	0.31	0.47	0.38
PART	0.51	0.49	0.50
DMNBtext	0.51	0.54	**0.53**
LibSVM	0.51	0.56	**0.53**
SMO	0.51	0.56	**0.53**
AdaBoostM1	0.51	0.57	**0.54**
BayesianLogisticRegression	0.49	0.59	**0.54**
NaiveBayesMultinomial	0.43	**0.76**	**0.55**
NaiveBayes	0.40	**0.77**	**0.53**

(Entrenamiento: Apache Commons - Pruebas: Java, JFace)

Tabla 6.7: Resultados de entrenar clasificadores con Java y probar sobre Apache Commons y JFace.

Clasificador	precision	recall	f-measure
ZeroR	0.00	0.00	0.00
OneR	0.48	0.09	0.16
IBk	0.62	0.23	0.34
RandomForest	0.71	0.43	0.53
Logistic	0.36	0.62	0.45
LibSVM	0.59	0.71	**0.64**
SMO	0.59	0.71	**0.64**
AdaBoostM1	0.56	0.71	0.63
BayesianLogisticRegression	0.55	0.73	0.63
DMNBtext	0.61	0.74	**0.67**
J48	0.59	0.79	**0.67**
NaiveBayes	0.50	**0.80**	0.61
NaiveBayesMultinomial	0.54	**0.80**	**0.64**
PART	0.54	**0.80**	**0.64**

(Entrenamiento: Java - Pruebas: Apache Commons, JFace)

Tabla 6.8: Resultados de entrenar clasificadores con JFace y probar sobre Apache Commons y Java.

Clasificador	precision	recall	f-measure
ZeroR	0.00	0.00	0.00
OneR	0.79	0.06	0.11
IBk	0.81	0.09	0.16
RandomForest	0.91	0.09	0.16
DMNBtext	0.84	0.21	0.34
J48	0.77	0.22	0.35
PART	0.72	0.28	0.41
AdaBoostM1	0.78	0.33	0.47
LibSVM	0.80	0.37	0.51
SMO	0.80	0.37	0.51
BayesianLogisticRegression	0.75	0.37	0.50
Logistic	0.61	0.46	**0.53**
NaiveBayes	0.68	**0.57**	**0.62**
NaiveBayesMultinomial	0.67	**0.68**	**0.68**

(Entrenamiento: JFace - Pruebas: Apache Commons, Java)

NaiveBayes y NaiveBayesMultinomial obtuvieron los mejores resultados de forma consistente para los tres casos del experimento. Para la primera y última tabla, los valores de *recall* y *F-Measure* alcanzados son regulares según *recall*, pero con Java en la segunda tabla, se obtuvo 0.8 de *recall* y 0.64 de *F-Measure*. Estos valores son suficientemente buenos según los objetivos buscados por este estudio, y quizás pueden ser mejorados usando matrices de costos.

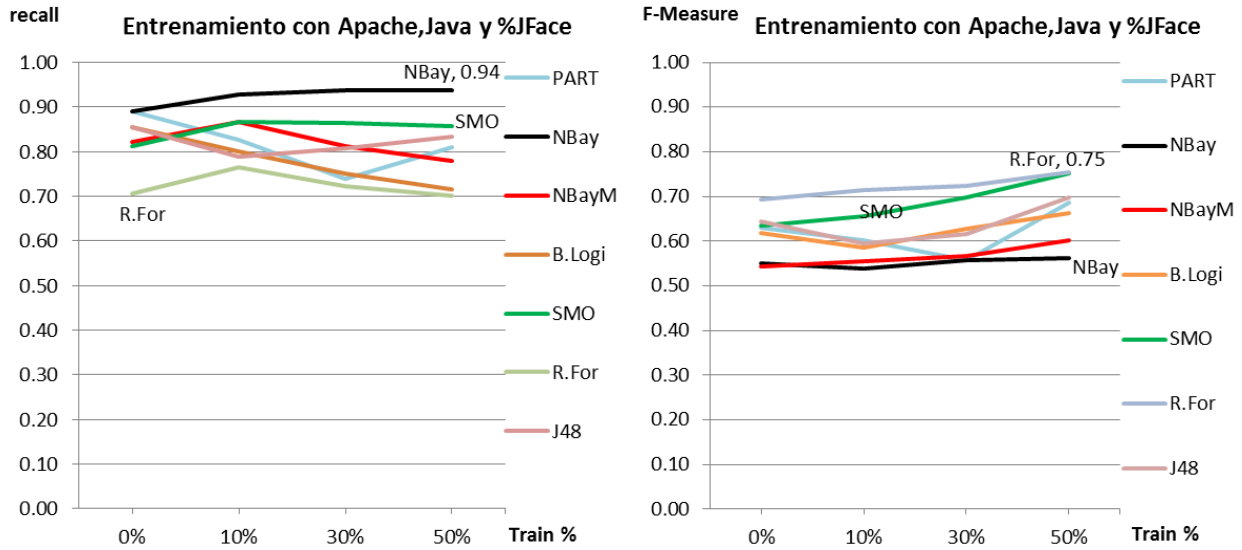
Una posible explicación del mejor rendimiento al usar Java para entrenar es que esta librería tiene una documentación más extensa y detallada en promedio. En efecto, al realizar una inspección manual de los comentarios de los datos finales, Java presenta mayor cantidad líneas de texto describiendo a los métodos en promedio. La diferencia es significativa, sobre todo con respecto a la API de Apache, la cual presenta la menor cantidad de líneas de descripción en la documentación. Puede ocurrir que la falta de documentación en Apache y en JFace tenga una repercusión negativa sobre el rendimiento de los clasificadores.

En la siguiente sección se muestran posiblemente mejores resultados al entrenar con 2 API y realizando pruebas con la API restante.

6.2.5 Entrenando con 2 API y probando con 1 API

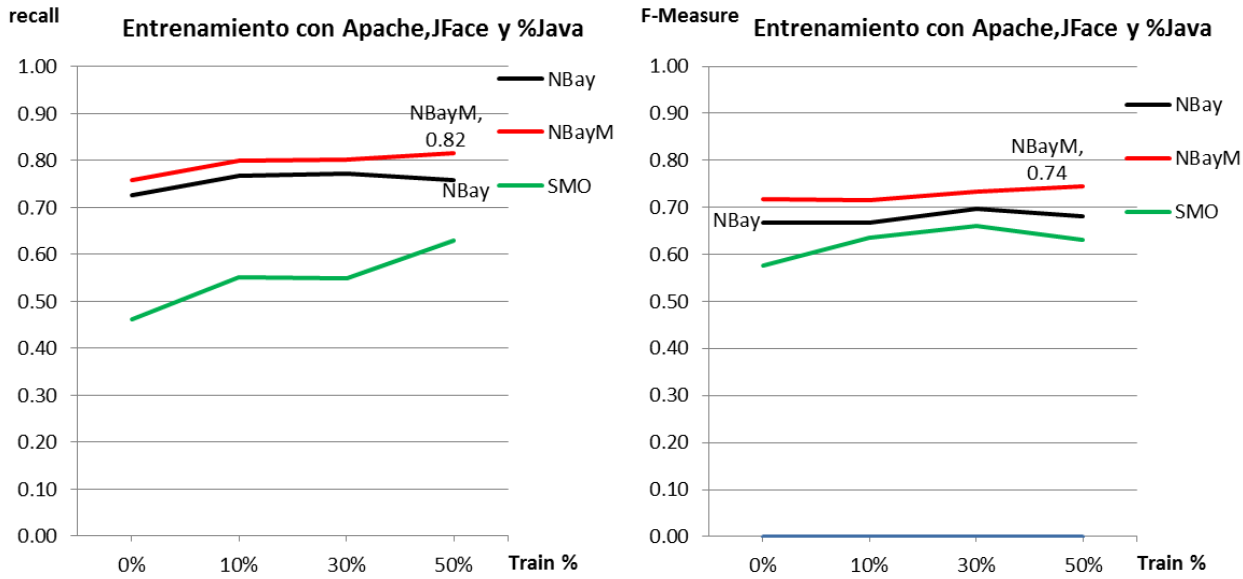
En estos experimentos se seleccionan 2 API junto con un porcentaje de la API restante, y se usa este conjunto de datos para entrenar, mientras que el porcentaje restante de la tercera API se usa para entrenar. Los resultados de la Imagen 6.2 corresponden a los clasificadores que fueron entrenados con Apache Commons, Java y un porcentaje de JFace y que fueron probados con la parte restante de JFace. En los gráficos, se omite el clasificador LibSVM por obtener resultados idénticos a SMO.

Imagen 6.2: Gráficos mostrando el los valores de recall y F-Measure para los clasificadores a medida que se aumenta la cantidad de datos considerados en la fase de entrenamiento. En este caso se usa como base de entrenamiento las librerías Apache Commons y Java.



Se observa que NaiveBayes tiene excelente *recall* y un *F-Measure* aceptable cercano a 0.5. SMO tiene un *recall* mayor a 0.85 desde un 10% de entrenamiento en adelante mientras que su *F-Measure* va aumentando al entrenar con más datos de JFace, llegando hasta 0.75 con un 50% de la API de JFace. Además una matriz de costos podría aplicarse a NaiveBayes y a SMO para mejorar aún más el *recall*. Los clasificadores con peor rendimiento (ZeroR, OneR, AdaBoostM1, DMNBtext, Ibk, Logistic) no fueron incluidos en los gráficos para mejorar la claridad de los gráficos, y LibSVM no fue incluido por mostrar resultados idénticos a SMO.

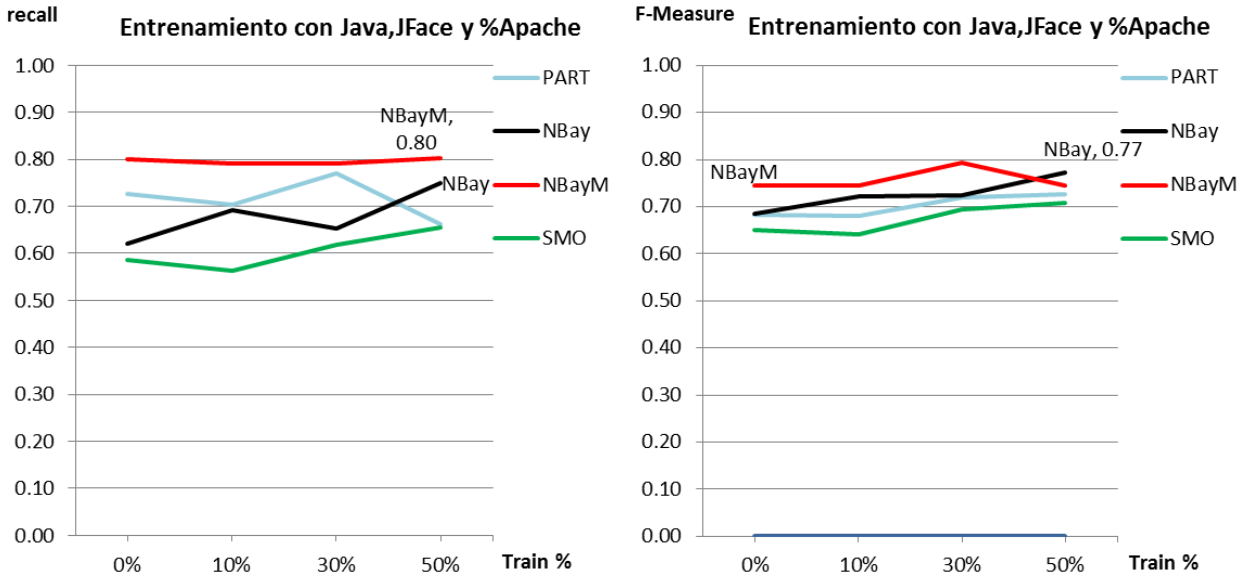
Imagen 6.3: Gráficos mostrando el los valores de recall y F-Measure para los clasificadores a medida que se aumenta la cantidad de datos considerados en la fase de entrenamiento. En este caso se usa como base de entrenamiento las librerías Apache Commons y JFace.



Los resultados de la Imagen 6.3 se obtuvieron entrenando con Apache Commons, JFace y un porcentaje de Java. El *test set* fue el porcentaje restante de Java. *NaiveBayesMultinomial* resultó

ser el con mejores resultados en todos los porcentajes, siguiéndole relativamente cerca *NaiveBayes*. Los otros clasificadores obtuvieron peores resultados (ZeroR, OneR, AdaBoostM1, DMNBtext, Ibk, Logistic, PART, BayesianLogisticRegresion, RandomForest, J48) por lo que no fueron incluidos en los gráficos nuevamente para mantener claridad. SMO fue incluido para comparar con los resultados anteriores donde sí tuvo un buen rendimiento y LibSVM fue omitido de los gráficos por obtener los mismos valores que SMO.

Imagen 6.4: Gráficos mostrando el los valores de recall y F-Measure para los clasificadores a medida que se aumenta la cantidad de datos considerados en la fase de entrenamiento. En este caso se usa como base de entrenamiento las librerías Java y JFace.



En la última combinación de los datos (Imagen 6.4) – entrenando con Java, JFace y con un porcentaje de Apache Commons – se alcanzaron mejores resultados nuevamente con NaiveBayesMultinomial, solo que esta vez presenta una caída del valor *F-Measure* del mismo clasificador cuando se entrena con el 50% de Apache. Esta vez NaiveBayes no comienza muy bien en 0%, pero mejora bastante cuando alcanza los 50%. Los clasificadores con peores resultados (ZeroR, OneR, AdaBoostM1, DMNBtext, Ibk, Logistic, BayesianLogisticRegresion, RandomForest, J48) no fueron incluidos en el gráfico para mejorar la claridad de las líneas, y LibSVM fue omitido por tener valores idénticos a SMO.

Estos experimentos apoyan a que los mejores clasificadores de directivas son NaiveBayes y NaiveBayesMultinomial, si se usan dos librerías completas para entrenar a los clasificadores. Además, el rendimiento de los clasificadores mejoró (la mayoría de los casos) a medida que se incluía un mayor porcentaje de la tercera API al conjunto de entrenamiento

Una observación es que, en el primer experimento, el mejor clasificador parece ser SMO porque presenta la mejor combinación entre *F-Measure* y *recall*. Pero NaiveBayes supera a SMO en los valores de *recall*, y es el valor de *recall* el que más importa según los objetivos de este estudio. Sin embargo, se puede especular que al aplicar una matriz de costos a SMO sería posible alcanzar el valor de 0.94 que tiene NaiveBayes en *recall* a los 50%. Pero al hacer esto probablemente bajaría el valor de *precision* de SMO hasta un valor cercano al *precision* de NaiveBayes, significando que ambos probablemente tuvieron un rendimiento similar. Como SMO sólo

presenta un buen rendimiento en el primer set de datos, es más conveniente seleccionar a NaiveBayes y NaiveBayesMultinomial como los mejores clasificadores para este experimento.

6.2.6 Reduciendo errores con la matriz de costos

Se puede aumentar el valor de *recall* de algunos clasificadores usando una matriz de costos, como se menciona en la sección 6.2.1. Esto se traduce en un mayor porcentaje de las directivas siendo correctamente detectadas, pero por otra parte un mayor porcentaje de no-directivas siendo incorrectamente clasificadas como directivas (*precision* más bajo). Sin embargo para los efectos de este trabajo se prefiere un mayor recall pues es más fácil corregir manualmente las no-directivas, como se describe en la sección 3.2.1.

En la Tabla 6.9 se ven los resultados de aplicar la matriz de costos. *NaiveBayes* logra mejorar el *recall* considerablemente en los tres conjuntos de datos probados mientras que *NaiveBayesMultinomial* no lo hace en ninguno de los datos. También hay que notar que en la última tabla entrenando con Java y JFace, el *recall* más alto se logra con un costo de 12 y con costos mayores el rendimiento empeoró.

Tabla 6.9: Efecto de la matriz de costos sobre NaiveBayes y NaiveBayesMultinomial. La primera tabla usa como entrenamiento a Apache Commons y Java; como pruebas usa a JFace. La segunda tabla usa como entrenamiento a Apache Commons y JFace; como pruebas usa a Java. La tercera y última tabla usa como entrenamiento a Java y JFace; como pruebas usa a Apache Commons.

	NaiveBayes			NaiveBayesMultinomial		
Costo FN	*precision*	*recall*	*F-Measure*	precision	recall	F-Measure
1	0.40	0.89	0.55	0.40	0.82	0.54
2	0.39	0.93	0.55	0.42	0.84	0.56
4	0.36	0.95	0.53	0.39	0.84	0.54
6	0.35	0.95	0.51	0.39	0.85	0.53
8	0.34	0.95	0.50	0.38	0.84	0.53
10	0.33	0.96	0.49	0.38	0.84	0.52
12	0.32	0.96	0.48	0.37	0.83	0.51
14	0.32	0.96	0.48	0.36	0.83	0.51
16	0.31	0.96	0.47	0.36	0.84	0.50
18	0.30	0.96	0.46	0.35	0.84	0.49
20	0.30	**0.98**	0.46	0.34	0.84	0.49

(Entrenamiento: Apache Commons, Java - Pruebas: JFace)

	NaiveBayes			NaiveBayesMultinomial		
Costo FN	*precision*	*recall*	*F-Measure*	precision	recall	F-Measure
1	0.62	0.73	0.67	0.68	0.76	0.72
2	0.60	0.77	0.67	0.69	0.70	0.69
4	0.56	0.81	0.66	0.66	0.62	0.64
6	0.54	0.82	0.65	0.63	0.59	0.61
8	0.54	0.84	0.66	0.61	0.56	0.59
10	0.53	0.85	0.65	0.59	0.54	0.56
12	0.52	0.86	0.65	0.57	0.52	0.54
14	0.51	0.87	0.64	0.55	0.51	0.53
16	0.51	0.87	0.64	0.53	0.51	0.52
18	0.51	0.88	0.64	0.52	0.51	0.52
20	0.50	**0.88**	0.64	0.51	0.51	0.51

(Entrenamiento: Apache Commons, JFace - Pruebas: Java)

	NaiveBayes			NaiveBayesMultinomial		
Costo FN	*precision*	*recall*	*F-Measure*	precision	recall	F-Measure
1	0.64	0.73	0.68	0.69	0.80	0.74
2	0.64	0.83	0.72	0.67	0.83	0.74
4	0.63	0.93	0.75	0.59	0.77	0.67
6	0.61	0.94	0.74	0.51	0.79	0.62
8	0.59	0.95	0.73	0.47	0.79	0.59
10	0.57	0.95	0.72	0.44	0.80	0.57
12	0.56	**0.96**	0.71	0.41	0.81	0.55
14	0.55	0.96	0.70	0.39	0.81	0.53
16	0.54	0.96	0.69	0.38	0.82	0.52
18	0.53	0.96	0.68	0.37	0.82	0.51
20	0.52	0.96	0.68	0.35	0.83	0.49

(Entrenamiento: Java, JFace - Pruebas: Apache Commons)

Usando la matriz de costos sobre NaiveBayes se pudo alcanzar más de 95% de correctitud detectando directivas. NaiveBayes puede ser mejorado en términos de *recall* usando este método lo cual lo deja como uno de los mejores candidatos para detectar la mayor cantidad de directivas de una API, al menos según los resultados de estos experimentos.

Por otra parte, con el clasificador *Logistic* también se realizaron pruebas usando la matriz de costos, pero usando los datos de la sección 6.2.3 – ‘Entrenando y probando con la misma API’.

Sin embargo los costos de matriz solo empeoraron el rendimiento de *Logistic* como lo muestra la Imagen 6.5. Los resultados de *Logistic* entrando sólo con un 2.5% de una API fueron buenos, pero inconsistentes. Puede ser un accidente que haya funcionado bien o también puede ser que sea un buen clasificador con porcentajes muy bajos de entrenamiento, lo cual puede ser un trabajo futuro a realizar para determinar el comportamiento de *Logistic* entrenando con porcentajes bajos.

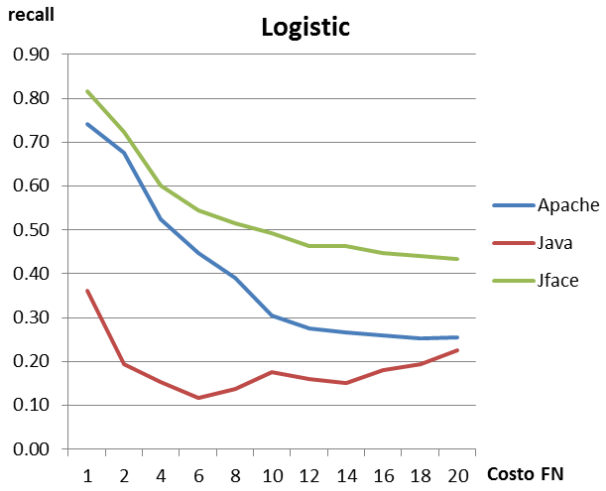


Imagen 6.5: Efecto de matriz de costos sobre *Logistic* entrenando con un 2.5% de cada API y realizando pruebas con el porcentaje restante.

6.2.7 Stop-words, stemming, TF-IDF Transform y *wordcounts*

En esta sección se describe un experimento diseñado para observar si se pueden conseguir mejores resultados aplicando listas de stop-words, stemming, transformación TF-IDF y wordcounts sobre los clasificadores NaiveBayes y NaiveBayesMultinomial. Antes de realizar el experimento, se realizan pruebas para seleccionar a la mejor lista de stop-words y al mejor algoritmo de stemming. Las pruebas son descritas a continuación y los resultados finales del experimento se pueden revisar en el Anexo - 9.2.

Usar stopwords puede ser útil para obtener mejores resultados en Text Mining. Consiste en ignorar palabras que usualmente aportan poca información semántica tales como 'and' o 'a'. Se realizó una prueba con los dos clasificadores más exitosos de los resultados anteriores comparando su rendimiento con dos listados de palabras stopwords. Un listado es uno publicado por la empresa Google[13] y el otro es una modificación de ese mismo, adaptado para el dominio de comentarios de API. La modificación consiste en considerar palabras que pueden importar en comentarios de librerías como 'may', 'this' o 'or' y se encuentra publicado en el repositorio de datos públicos de este trabajo (ver sección 4.1).

La Tabla 6.10 muestra el rendimiento de *NaiveBayes* y N*aiveBayesMultinomial* cuando se les aplica una lista de *stop-words* al proceso de separación de palabras que realiza el filtro *StringToWordVector*. Se está comparando entre dos listas de stop-words para seleccionar la que causa mejores resultados. En la tabla se ve que sin usar stop-words se obtienen los mejores resultados. Pero no se sabe aún cómo es el comportamiento de los clasificadores si combinamos stop-words con stemming y TF-IDF, por lo que por ahora se selecciona la mejor lista de stop-

[13] Google stopwords en https://code.google.com/p/stop-words/

words. Esta lista es usada en el experimento descrito en esta sección que tiene los resultados en el Anexo - 9.2.

Sin embargo, las listas de stop-words presentan resultados muy similares entre sí y al parecer no afectaría cuál de las dos listas es usada. De esta forma, se decide seleccionar la lista *stop-words* de Google. Las combinaciones con wordcounts fueron hechas pensando que el no usarlas podría significar un mal rendimiento para NaiveBayesMultinomial, el cual se conoce por tener un mejor comportamiento al usar este parámetro. Wordcounts trabaja con la frecuencia de cada palabra en una frase, a diferencia de utilizar *{verdadero,falso}* para indicar la presencia de cada palabra. Curiosamente, el uso de wordcounts empeoró el rendimiento en estos resultados para ambos clasificadores.

Tabla 6.10: Comparación de rendimiento al usar dos listas de stopwords. La primera fila de resultados corresponde al caso donde no se usó stop-words ni wordcounts.

		NaiveBayes		NaiveBayesMultinomial	
Stop-words	Wordcounts	recall	f-measure	recall	f-measure
-	-	**0.862**	0.375	**0.848**	**0.475**
google	-	0.804	**0.396**	**0.833**	**0.478**
google	si	0.761	0.413	0.826	0.463
google-modificado	-	**0.812**	0.413	0.833	0.470
google-modificado	si	0.717	**0.419**	0.819	0.453

Por otra parte se comparó a tres algoritmos de stemming, o *stemmers*, con los clasificadores NaiveBayes y NaiveBayesMultinomial, para observar si se consiguen mejores resultados. En la Tabla 6.11 se muestra que no se obtuvieron mejoras al aplicar stemming, y entre los tipos de stemming el mejor resultó ser *Snowball*, por lo cual es seleccionado como el mejor stemmer del experimento.

Tabla 6.11: Comparación de rendimiento al usar distintos stemmers. La primera fila de resultados corresponde al caso donde no se realizó stemming.

	NaiveBayes		NaiveBayesMultinomial	
Stemmer	recall	f-measure	recall	f-measure
-	**0.780**	**0.585**	**0.590**	**0.655**
Lovins	0.772	0.566	0.569	0.635
IteratedLovins	0.774	0.565	0.553	0.619
Snowball	**0.780**	**0.585**	**0.590**	**0.655**

El objetivo de este estos experimentos es determinar si existe alguna combinación que mejore el rendimiento de *NaiveBayes* o de *NaiveBayesMultinomial*. Ya fueron escogidas a la lista de Google y a Snowball como mejores stop-words y stemmers respectivamente. Otro parámetro que podría mejorar el rendimiento de los clasificadores es la aplicación de TF-IDF. Las

transformaciones TF-IDF ayudan a considerar el peso de las palabras comunes versus las palabras infrecuentes. De esta manera, se realiza un experimento para comparar el rendimiento al aplicar u omitir el uso de stop-words, stemming y TF-IDF. Los resultados del experimento con las 16 combinaciones se encuentran en el Anexo - 9.2.

En resumen no se observaron mejoras considerables como para concluir que es mejor usar alguno de esas combinaciones nuevas. Al entrenar con Apache Commons y Java funcionó mejor *NaiveBayesMultinomial* sin opciones extra, al entrenar con Apache Commons y JFace funcionó mejor *NaiveBayes* sin opciones extra y al entrenar con Java y JFace el mejor fue *NaiveBayes* con stopwords. Es decir, no se obtuvieron resultados que apoyaran el uso de stemming, stop-words o wordcounts.

6.2.8 Tiempos de entrenamiento y de prueba

La Imagen 6.6 muestra el tiempo en segundos que tarda cada clasificador considerado en entrenar y realizar pruebas con 2951 instancias – los gráficos están con distintas escalas en el eje vertical. Esta información puede ser útil al momento de elegir entre dos clasificadores con rendimientos muy similares, eligiendo por supuesto al más rápido. Los tiempos fueron obtenidos usando el Experimenter de Weka.

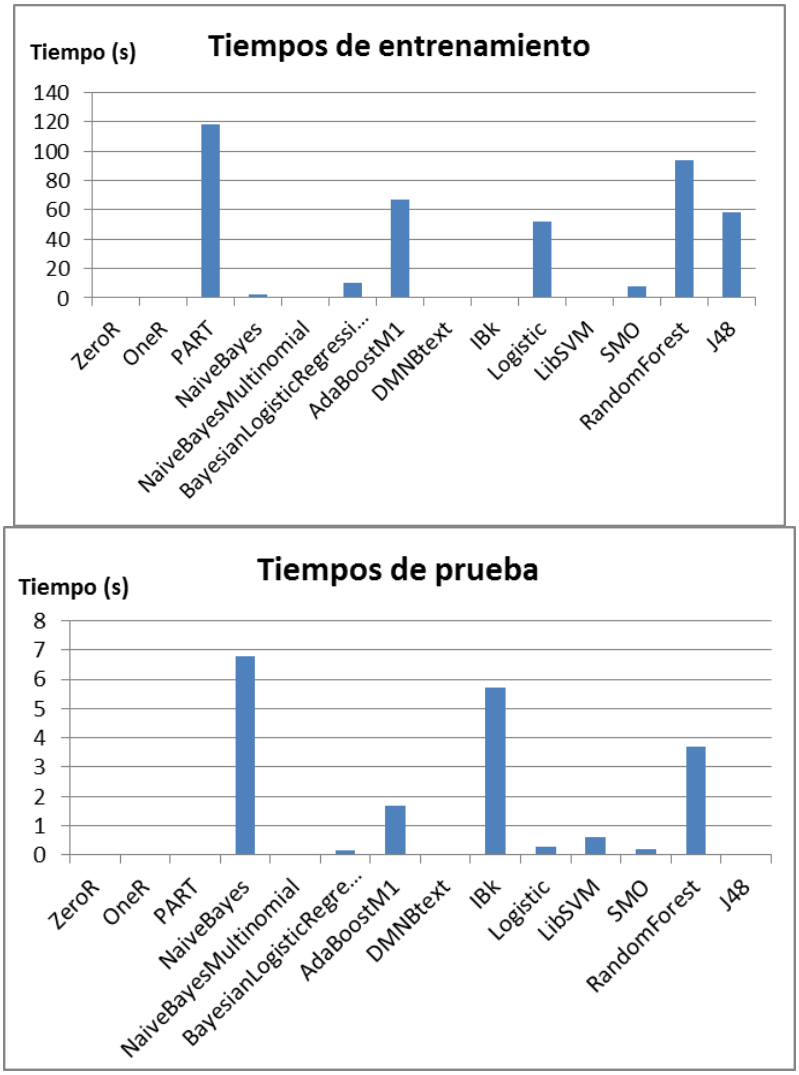


Imagen 6.6: Tiempos de entrenamiento y prueba de cada clasificador sobre 2951 instancias al entrenar y 2952 instancias en las pruebas

7 Conclusiones y trabajo futuro

7.1 Conclusiones

Una conclusión evidente de los resultados es que efectivamente si se pueden detectar las directivas con un grado satisfactorio de precisión y exactitud usando algoritmos de aprendizaje de máquina. Los resultados apoyan que usando clasificadores de *Machine Learning* se pueden lograr mejores rendimientos detectando directivas que usando la regla de los *keywords* (una frase es una directiva si contiene alguno de los keywords).

Según los resultados sobre los datos finales obtenidos se observa lo siguiente:

- Entrenar clasificadores con un porcentaje menor a 30% de una API, sin tener un modelo previo basado en otras librerías, alcanza a obtener resultados regulares
- Entrenar clasificadores con 1 API para formar un modelo base puede obtener resultados decentes y buenos detectando directivas de otras librerías.
- Entrenar clasificadores con 2 API, formando un modelo base, puede mejores resultados que entrenar con 1 API, alcanzando un buen rendimiento al detectar directivas.

Pero al entrenar con 1 API se realizaron pruebas sobre 2 API. Mientras que al entrenar con 2 API, las pruebas se efectuaron sobre 1 API. De este modo, es complicado concluir que al entrenar con más librerías se obtienen mejores detecciones de directivas, pues las condiciones de comparación entre ambos experimentos son distintas.

Los datos revelan que los mejores clasificadores, sobre los datos finales considerados en este trabajo, son los siguientes:

- NaiveBayes: Logró en general los mejores resultados y fue suficientemente consistente a lo largo de los experimentos. Además presenta un buen comportamiento con el uso de la matriz de costos, lo cual le permite aumentar la cantidad de directivas detectas por el costo de aumentar la cantidad de falsos positivos. Esto último permite ajustar a NaiveBayes hasta que alcance el porcentaje de directivas que uno necesita detectar. Es además un algoritmo rápido en comparación a otros clasificadores. Sin embargo hay casos donde NaiveBayes no tuvo un buen rendimiento, pero por otra parte esto sucedió con todos los otros clasificadores.
- NaiveBayesMultinomial: Logró resultados comparables o mejores que NaiveBayes, sin embargo fue algo más inconsistente pues en algunos experimentos su rendimiento fue muy pobre. Un problema de este clasificador es que no parece poder mejorar su rendimiento cuando se le aplica una matriz de costos, haciéndola menos versátil que NaiveBayes.
- SMO: Este clasificador suele comenzar a presentar buenos resultados al alcanzar entrenamientos mayores a 30% o a veces 50% pareciendo superar el rendimiento de los algoritmos Bayesianos, pero no presenta un buen comportamiento en porcentajes bajos de entrenamiento por lo general. LibSVM presentó resultados idénticos a SMO en casi todos los experimentos, revelando que quizás pueden compartir una implementación muy similar.

Por otra parte, los experimentos realizados con los datos preliminares son menos representativos de un caso real donde es necesario detectar directivas sobre una API completamente desconocida que probablemente ha sido documentada con otro estilo y distintas palabras. Los experimentos con los datos preliminares entrenan con un porcentaje mayor al 90% de un extracto de una API y luego realizan pruebas sobre 200 frases obtenidas de exactamente la misma muestra. Los

resultados con estos datos favorecieron a SMO y a J48. Pero por las razones explicadas, los resultados con los datos preliminares no son considerados como representativos de un escenario real con directivas sobre librerías desconocidas.

Además se descubrieron las siguientes características de algunos clasificadores:

- El clasificador LogisticRegresion no mejora al cambiar su matriz de costos.
- No se observaron mejoras en NaiveBayes ni NaiveBayesMultinomial al probar el uso de stopwords, stemming, wordcounts ni TF-IDF Transform.

7.2 Trabajo Futuro

Como trabajo futuro puede ser interesante estudiar el rendimiento de clasificadores considerando una tercera clase *directiva-null*, la cual considera a las directivas relacionadas con el valor null de Java.

La aplicación CHi puede ser extendida con la opción de poder usar más clasificadores de los que tiene actualmente. Además podría poseer clasificadores incrementales, los cuales continúan entrenándose automáticamente con datos nuevos. Esto permitiría seguir mejorando su rendimiento a medida que avanza el tiempo, formando un modelo dinámico de clasificación.

Se podría considerar otra arista de avances investigando la posibilidad de destacar directivas adentro de editores de texto y adentro de documentación web de API, ayudando a automatizar más el sistema para detectar directivas.

Hay varias otras tareas que se pueden realizar como trabajos futuros relacionados a este estudio:

- Probar más combinaciones en el filtro StringToWordVector tales como wordsToKeep, que indica cuántas palabras podrá analizar un clasificador, como *tokenizer*, probar stemmers externos a Weka, mejorar la lista de *stopwords*, eliminar manualmente palabras del conjunto de palabras consideradas, etc.
- Mejorar la estrategia que detecta directivas usando keywords. Quizás un mejor conjunto de palabras clave puede superar el rendimiento de los clasificadores de Text Mining.
- Aplicar herramientas de Text Mining para seleccionar atributos (provisto por Weka), lo cual podría servir para obtener una lista de palabras que son características del dominio del problema. Comparar estos resultados con la lista de keywords.
- Analizar los nodos superiores del árbol que genera el clasificador J48. Estos pueden entregar información valiosa sobre las palabras claves que influyen de mayor manera en la clasificación de directivas y no-directivas.
- Aplicar herramientas de Text Mining para detectar *clusters;* conjuntos que agrupan instancias similares entre sí (provisto por Weka). Esto podría permitir conocer más sobre la naturaleza del problema de clasificar directivas de API.
- Probar rendimiento de clasificadores disponibles en la librería opensource libLinear, la cual tiene buena reputación en el área de clasificación de documentos. [2]
- Probar entrando clasificadores con otras API distintas a Apache Commons, Java y JFace.

8 Bibliografía

[1] D. Steidl, B. Hummel, E. Juergens "Quality Analysis of Source Code Comments," 2013.

[2] Clasificación de Documentos usando libLinear (Apéndice C): http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf.

[3] U. Dekel, J.D. Herbsleb "Reading the Documentation of Invoked API Functions in Program Comprehension," ICPC, 2009.

[4] E. Soloway, R. Lampert, S. Letovsky, D. Littman, and J. Pinto "Designing documentation to compensate for delocalized plans," ACM, 31(11):1259–1267, 1988.

[5] U. Dekel, J.D. Herbsleb "Improving API Documentation Usability with Knowledge Pushing," 2009.

[6] M. Monperrus, M. Eichberg, E. Tekes, M. Mezini "What should developers be aware of? An empirical study on the directives of API documentation," 2011.

9 Anexo

9.1 Pruebas extras con NaiveBayes y SMO (datos preliminares)

Durante los experimentos con los datos preliminares se realizaron algunas pruebas extras, probando si existían diferencias al variar la lista de stop-words y algunos parámetros del clasificador SMO. En estos resultados no se registraron los valores de *recall* ni *precision*; sólo fueron registrados los porcentajes de frases correctamente clasificadas, considerando ambas clases, *directiva* y *no-directiva*.

Tabla 9.1: Porcentaje de frases correctamente clasificadas sobre 200 instancias de los datos preliminares usando diferentes listas de stop-words con NaiveBayes y SMO.

Stop-words	NaiveBayes	SMO
-	78.0%	95.5%
stop-words-english1	78.5%	93.5%
stop-words-english2	77.5%	95.5%
stop-words-english3-google	79.5%	94.0%
stop-words-english4	77.5%	93.5%
stop-words-english5	77.5%	93.5%

Tabla 9.2: Porcentaje de frases correctamente clasificadas sobre 200 instancias de los datos preliminares usando el clasificador SMO. Los parámetros por defecto, que fueron usados durante los demás experimentos del estudio, son c=1 y kernel=Poly Kernel, por lo tanto en esta tabla el valor 95.5% con c=1 y kernel=Poly Kernel corresponde al rendimiento sin variar el parámetro c de SMO ni el tipo de Kernel.

c	poly kernel	Normalized Poly kernel	PUK
1	95.5%	91.5%	88.0%
0.85	95.5%	-	-
0.4	96.5%	-	-
0.2	97.5%	-	-
0.05	96.0%	-	-

9.2 Stop-words, stemming, TF-IDF Transform y wordcounts (datos finales)

Distintas combinaciones de parámetros con el fin de buscar una mejora en el rendimiento de *NaiveBayes* o de *NaiveBayesMultinomial.* Son tres experimentos, tomando dos API para entrenar los clasificadores y probando los modelos resultantes con la API restante. Los mejores valores se encuentran destacados.

				NaiveBayes			NaiveBayesMultinomial		
TF-IDF Transform	Word counts	Stopwords	Stemming	*precision*	*recall*	*F-Measure*	precision	recall	F-Measure
-	-	-	-	0.239	**0.862**	0.375	**0.330**	**0.848**	0.475
-	-	-	si	0.239	0.862	0.375	0.330	0.848	0.475
-	-	si	-	0.263	0.804	0.396	0.335	0.833	**0.478**
-	-	si	si	0.263	0.804	0.396	0.335	0.833	0.478
-	si	-	-	0.271	0.797	0.404	0.320	0.841	0.464
-	si	-	si	0.271	0.797	0.404	0.320	0.841	0.464
-	si	si	-	**0.284**	0.761	**0.413**	0.322	0.826	0.463
-	si	si	si	0.284	0.761	0.413	0.322	0.826	0.463
si	-	-	-	0.239	0.862	0.375	0.191	0.841	0.311
si	-	-	si	0.239	0.862	0.375	0.191	0.841	0.311
si	-	si	-	0.263	0.804	0.396	0.185	0.797	0.300
si	-	si	si	0.263	0.804	0.396	0.185	0.797	0.300
si	si	-	-	0.208	0.739	0.324	0.192	**0.855**	0.313
si	si	-	si	0.208	0.739	0.324	0.192	0.855	0.313
si	si	si	-	0.240	0.703	0.357	0.183	0.790	0.297
si	si	si	si	0.240	0.703	0.357	0.183	0.790	0.297

(Entrenamiento: Apache Commons, Java - Pruebas: JFace)

				NaiveBayes			NaiveBayesMultinomial		
TF-IDF Transform	Word counts	Stopwords	Stemming	*precision*	*recall*	*F-Measure*	precision	recall	F-Measure
-	-	-	-	0.618	**0.726**	0.668	**0.760**	0.531	0.625
-	-	-	si	0.618	0.726	0.668	0.760	0.531	0.625
-	-	si	-	**0.688**	0.655	**0.671**	**0.762**	0.535	0.629
-	-	si	si	0.688	0.655	0.671	0.762	0.535	0.629
-	si	-	-	0.617	0.707	0.659	0.736	0.538	0.622
-	si	-	si	0.617	0.707	0.659	0.736	0.538	0.622
-	si	si	-	0.668	0.625	0.646	0.751	0.546	**0.632**

-	si	si	si	0.668	0.625	0.646	0.751	0.546	0.632
si	-	-	-	0.618	0.726	0.668	0.589	**0.590**	0.590
si	-	-	si	0.618	0.726	0.668	0.589	0.590	0.590
si	-	si	-	0.688	0.655	0.671	0.579	0.589	0.584
si	-	si	si	0.688	0.655	0.671	0.579	0.589	0.584
si	si	-	-	0.582	0.719	0.643	0.587	0.587	0.587
si	si	-	si	0.582	0.719	0.643	0.587	0.587	0.587
si	si	si	-	0.637	0.614	0.626	0.579	0.590	0.585
si	si	si	si	0.637	0.614	0.626	0.579	0.590	0.585

(Entrenamiento: Apache Commons, JFace - Pruebas: Java)

				NaiveBayes			NaiveBayesMultinomial		
TF-IDF Transform	Word counts	Stopwords	Stemming	*precision*	*recall*	*F-Measure*	precision	recall	F-Measure
-	-	-	-	0.644	0.726	0.683	**0.717**	0.774	**0.745**
-	-	-	si	0.644	0.726	0.683	**0.717**	0.774	**0.745**
-	-	si	-	**0.656**	**0.856**	**0.743**	0.698	0.784	0.739
-	-	si	si	**0.656**	**0.856**	**0.743**	0.698	0.784	0.739
-	si	-	-	0.507	0.510	0.509	0.697	0.788	0.740
-	si	-	si	0.507	0.510	0.509	0.697	0.788	0.740
-	si	si	-	0.527	0.678	0.593	0.680	0.719	0.699
-	si	si	si	0.527	0.678	0.593	0.680	0.719	0.699
si	-	-	-	0.644	0.726	0.683	0.561	0.822	0.667
si	-	-	si	0.644	0.726	0.683	0.561	0.822	0.667
si	-	si	-	**0.656**	**0.856**	**0.743**	0.556	0.812	0.660
si	-	si	si	**0.656**	**0.856**	**0.743**	0.556	0.812	0.660
si	si	-	-	0.396	0.747	0.517	0.538	0.822	0.650
si	si	-	si	0.396	0.747	0.517	0.538	0.822	0.650
si	si	si	-	0.389	0.781	0.519	0.550	**0.825**	0.660
si	si	si	si	0.389	0.781	0.519	0.550	**0.825**	0.660

(Entrenamiento: Java, JFace - Pruebas: Apache Commons)