

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 CORREA GAETE

PROFESOR GUÍA:

ROMAIN ROBBES

MIEMBROS DE LA COMISIÓN:

JORGE PÉREZ R.

BÁRBARA POBLETE L.

SANTIAGO DE CHILE

2015

Índice

[1 Introducción 1](#_Toc428443069)

[1.1 Documentación de API 1](#_Toc428443070)

[1.2 Problema con las Directivas 2](#_Toc428443071)

[Una Solución 3](#_Toc428443073)

[1.3 3](#_Toc428443074)

[1.4 Objetivos 3](#_Toc428443076)

[1.4.1 Objetivos Generales 3](#_Toc428443078)

[1.4.2 Objetivos Específicos 3](#_Toc428443079)

[2 Marco teórico y trabajos anteriores 4](#_Toc428443081)

[2.1 Importancia de comentarios en API y código fuente 4](#_Toc428443082)

[2.2 Destacación de directivas en API 4](#_Toc428443083)

[2.2.1 Problema: Las directivas pueden pasar desapercibidas 4](#_Toc428443084)

[2.2.2 Syntax highlighting en editores de texto 4](#_Toc428443085)

[2.2.3 eMoose y tagSea 4](#_Toc428443086)

[2.3 Trabajo de Monperrus et al. 4](#_Toc428443087)

[2.3.1 Keywords para detectar directivas 4](#_Toc428443088)

[2.3.2 Taxonomía de directivas 4](#_Toc428443089)

[3 Desarrollo de <NombrePrograma> 5](#_Toc428443090)

[3.1 Usuarios Finales 5](#_Toc428443091)

[3.1.1 Usuario Encargado de Documentar una API 5](#_Toc428443092)

[3.1.2 Usuario Investigador de Text Mining 5](#_Toc428443093)

[3.2 Casos de Uso 6](#_Toc428443094)

[3.2.1 Documentador de API 6](#_Toc428443095)

[3.2.2 Investigador de Text Mining 6](#_Toc428443096)

[3.3 Interfaz de <NombrePrograma> 7](#_Toc428443097)

[3.4 Implementación de <NombrePrograma> 8](#_Toc428443098)

[3.5 Trabajo Futuro para <NombrePrograma> 8](#_Toc428443099)

[Revisión Manual de Comentarios de API 9](#_Toc428443100)

[4 9](#_Toc428443103)

[4.1 Revisión de los datos preliminares 9](#_Toc428443104)

[4.1.1 Método de Revisión de Datos Preliminares 10](#_Toc428443105)

[4.1.2 Observaciones sobre los datos preliminares 10](#_Toc428443106)

[4.2 Revisión de los datos finales 12](#_Toc428443107)

[4.2.1 Método de revisión de los datos finales 12](#_Toc428443108)

[5 Aplicación de Text Mining 13](#_Toc428443109)

[5.1 Weka 13](#_Toc428443110)

[5.1.1 Filtros 13](#_Toc428443111)

[5.2 Clasificadores y parámetros probados 15](#_Toc428443112)

[5.3 Evaluación del rendimiento de un clasificador 16](#_Toc428443113)

[6 Resultados 22](#_Toc428443115)

[6.1 Resultados Preliminares 22](#_Toc428443116)

[Resultados Finales 25](#_Toc428443118)

[6.2 25](#_Toc428443122)

[6.2.1 F-Measure vs Recall 25](#_Toc428443123)

[6.2.2 Usando Keywords 27](#_Toc428443124)

[6.2.3 Usando una API 28](#_Toc428443125)

[6.2.4 Entrenando con 1 API y probando con 2 API 29](#_Toc428443126)

[6.2.5 Entrenando con 2 API y probando con 1 API 31](#_Toc428443127)

[6.2.6 Reduciendo errores con la matriz de costos 33](#_Toc428443128)

[6.2.7](#_Toc428443129) *[Stop-words, stemming,](#_Toc428443129)* [TF-IDF Transform y](#_Toc428443129) *[wordcounts](#_Toc428443129)* [35](#_Toc428443129)

[6.2.8 Tiempos de entrenamiento y de prueba 36](#_Toc428443133)

[7 Discusión general 37](#_Toc428443135)

[8 Limitaciones y problemas de validez (?) 37](#_Toc428443136)

[9 Conclusiones y trabajo futuro 37](#_Toc428443137)

[10 Glosario 37](#_Toc428443138)

[11 Bibliografía 38](#_Toc428443139)

[12 Anexo 39](#_Toc428443140)

[12.1 Stop-words, stemming, TF-IDF Transform y wordcounts 39](#_Toc428443141)

# Introducción

## Documentación de API

En la actualidad existe una amplia variedad de programas y herramientas disponibles en el mercado de software. Estos programas son usados por distintos usuarios que deben conocer cómo usar tales herramientas lo cual normalmente significa pasar por una etapa de aprendizaje al 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 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. Es así como los manuales que explican el uso una API son frecuentemente visitados por los desarrolladores de software. Estos manuales son comúnmente llamados documentación de API y son generalmente provistos en la web del creador de cada librería.

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 **¡Error! No se encuentra el origen de la referencia.** muestra un ejemplo de documentación de la librería *Ant* de Java donde se escribió un texto explicativo antes del código de una función.

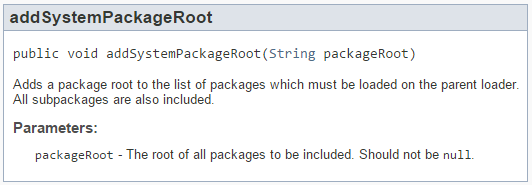
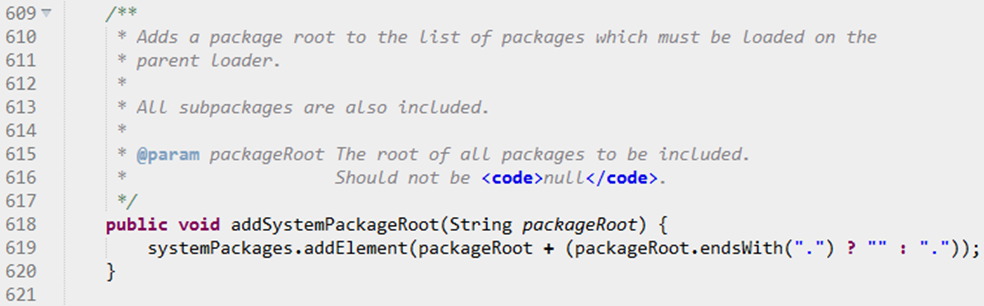


Imagen 1.1: Ejemplo de documentación en Java de tipo Javadoc. Arriba aparece el comentario adentro del archivo de código y abajo se encuentra el mismo comentario en forma de HTML después de ser procesado por Javadoc.

## Problema con las Directivas

En todo manual de instrucciones pueden existir sugerencias críticas, o al menos más importantes que otros consejos de uso, como por ejemplo la instrucción de no encender un microondas con metales 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. Lo siguiente se entiende por una directiva durante este trabajo:

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 como lo muestra la Imagen 1.2 donde hay una instrucción importante entremedio de otras frases. Todo esto se traduce a un probable aumento de errores de código, tiempo solucionando *bugs* y problemas de mantenimiento del código [[1](" \l "UDe09_2)].

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. [[2](" \l "ESo88)]

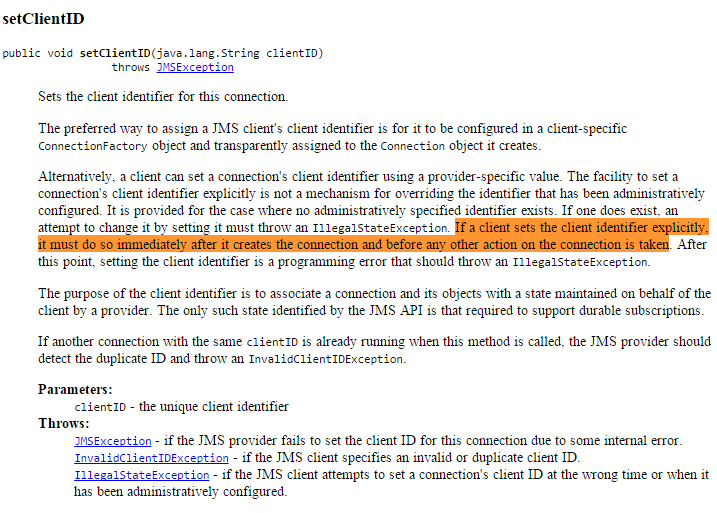


Imagen 1.2: Documentación de un método de Java JMS. La frase destacada es una directiva que no es simple de detectar a primera vista pues se encuentra al medio de un largo párrafo.  
*(en la documentación original la directiva no aparece destacada)*

## Una Solución

Para ayudar a que las directivas sean más fácilmente detectables se podría obligar a seguir un procedimiento estándar al crear la documentación de una API o bien se podría destacar de alguna manera las directivas. La primera opción probablemente tardaría años en adoptarse y puede ser complicado lograr que sea usado en todo proyecto de software. Por otra parte se pueden realizar esfuerzos para destacar visualmente las directivas en la documentación.

Sin embargo para destacar las directivas primero es necesario detectar cuáles son las instrucciones importantes de la documentación. Sin embargo una librería puede tener demasiados comentarios como para ser revisados por una persona. Por ejemplo, Java tiene más de 4000 clases, cada una con varios métodos comentados. Una revisión manual puede ser muy tediosa y además costosa en horas-hombre para una empresa.

Una solución – la propuesta por este trabajo – es que se detecten de manera semi-automática las directivas de API usando un clasificador entrenado por *Machine Learning* y herramientas de *Text Mining*, aprendiendo de comentarios en de librerías API como fuente. Es decir, un programa adivina mayor cantidad de directivas posibles y luego opcionalmente se puede revisar de forma manual para corregir los 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.

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

## Objetivos

### 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.

### 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.

# Marco teórico y trabajos anteriores

## Importancia de comentarios en API y código fuente

* % de comentarios en código fuente
* Trabajos pasados que apoyan la inclusión de comentarios para ayudar a desarrolladores

## Destacación de directivas en API

### Problema: Las directivas pueden pasar desapercibidas

* Mostrar ejemplos de directivas no vistas

### Syntax highlighting en editores de texto

* Mencionar que es posible extender eclipse y sublime text para que haga highlight de ciertas cosas.. que esto podría ayudar a destacar directivas adentro de los editores.
* Mostrar imágenes de la prueba realizada en sublime text

### eMoose y tagSea

* Resultados de este estudio con las directivas destacadas en Eclipse
* Imágenes de eMoose
* Explicación de eMoose y su posible utilidad y poder
* Taxonomía propuesta

## Trabajo de Monperrus et al.

[[Mencionar que el trabajo presenta una buena referencia de comparación al detectar directivas automáticamente, pues el trabajo obtuvo los porcentajes de keywords en cada directiva que encontraron]]

Monperrus et al. [[\*\*]] 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 ‘ecouraged’.

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

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

Juntando todas estas palabras claves, a las cuales nos referiremos a ellas 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 *‘inherit’*.

[[ (hablar de la comparación base de clasificadorDeKeywords)

]]

### Keywords para detectar directivas

### Taxonomía de directivas

# Desarrollo de <NombrePrograma>

Nombres posibles: Comments Highlighter (CHI o CHi); Java Api Highlighter (JAH); Jah Api Highlighter (JAH)

<NombrePrograma> 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. Resultó ser algo necesario para facilitar y hacer menos tediosa la 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 esfuerzo concluyó en un software que va a poder facilitar futuras revisiónes 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 <NombrePrograma> 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. <NombrePrograma> ayuda a destacar directivas dentro de los comentarios de una API.

## Usuarios Finales

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

### Usuario Encargado de Documentar una API

Es una persona que está documentando una librería con javadoc y le interesa destacar las directivas de su API. Quiere que los usuarios que lean su documentación puedan advertir con mayor facilidad las instrucciones importantes sobre el uso correcto de la librería. Entonces puede usar el programa para que las directivas sean detectadas automáticamente. Luego si desea puede revisar y corregir manualmente las directivas que fueron detectadas usando la aplicación.

Este tipo de usuario documenta potencialmente una librería grande con varias clases y comentarios. Revisar manualmente todos los comentarios en búsqueda de las directivas es un proceso demasiado largo, pero con <NombrePrograma> las gran mayoría de las directivas son destacadas automáticamente. Luego el usuario puede revisar las frases destacadas porque algunas de ellas pueden ser realmente comentarios normales. Se asume que el usuario necesita que el programa pueda detectar más del 85% de las directivas.

### 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 esto que este tipo de usuario puede tener interés en utilizar <NombrePrograma>.

## Casos de Uso

### 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[[1]](#footnote-1) 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 <NombrePrograma> suele equivocarse poco en esto.

### 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.

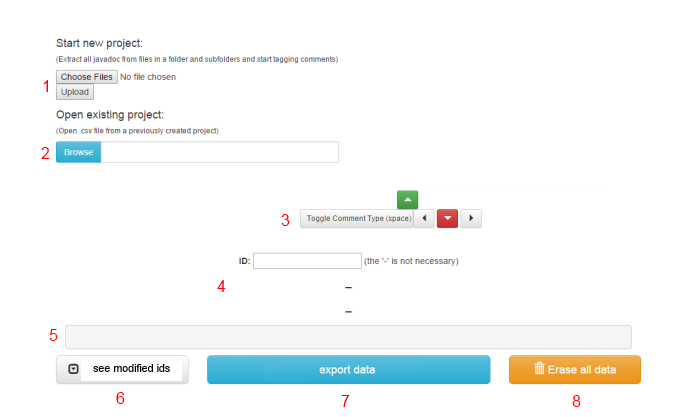
Luego de seleccionar la carpeta, la página toma unos segundos en procesar los archivos y luego muestra el primer comentario de API, indicando que el proyecto ya ha sido creado y está listo para ser manualmente revisado.

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.

## Interfaz de <NombrePrograma>

En la siguiente imagen se muestra la interfaz de <NombrePrograma> y luego se detalla cada componente y funcionalidad.



(esta imagen va a ser actualizada y mejorada después)

1. Comenzar nuevo proyecto

Selecciona la carpeta que contiene tu código fuente Java.

1. Abrir proyecto

Continua revisando un proyecto ya existente.

1. Navegación con botones (se recomienda usar hotkeys)

Anda al anterior o siguiente comentario/frase y destaca una frase. Se recomienda usar las teclas de acceso rápido (teclas de flechas y tecla de espacio).

1. Información

ID y path del comentario actual.

1. Visualización de comentarios

Zona donde se despliegan los comentarios.

1. Ver IDs modificados

Ver lista de IDs modificados hasta ahora. Puede ser útil en ciertas circunstancias, como por ejemplo para saber cuál fue la última ID que revisaste.

1. Exportar proyecto

Exportar las marcas manualmente realizadas a .csv y .arff.

1. Borrar datos

Borrar todas las marcas manualmente realizadas.

## Implementación de <NombrePrograma>

* Extracción de comentarios de varios archivos en una carpeta
* Mostrar capacidades de la página, imágenes, input, output, hablar de importancia de hotkeys, hablar de dificultades encontradas para lograr una buena visualización de comentarios
* Visualización de comentarios
* Extracción de comentarios
* Pre-proceso de comentarios
* Exportación de resultados
* Detalles de cliente-servidor

## Trabajo Futuro para <NombrePrograma>

* Extensibilidad a otros lenguajes
* Extensibilidad a más clases de comentarios non-directive,directive,…
* Desarrollo Futuro e Ideas Propuestas (ej. Insertar directivas en el javadoc original pasado por el usuario)
* Falta insertar directivas en el texto original

# Revisión Manual de Comentarios de API

Fue necesario realizar una revisión manual de comentarios para obtener datos requeridos por las herramientas de Machine Learning. Es decir, fue para que el programa tuviera ejemplos con los cuales pudiera 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*.

Todos los comentarios fueron extraídos del código fuente de JFace, Apache Commons y Java, los cuales se extrajo un total de 71603 frases siendo Java la librería con mayor cantidad de frases (45270) y Apache Commons la librería con menor cantidad de frases (7101).

La revisión manual de comentarios fue realizada usando <NombrePrograma>. 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 con conocimientos en el tema de directivas y documentación de API. Cada comentario fue leído y asignado por los revisadores de forma separada e independiente y los resultados fueron comparados posteriormente. Las diferencias de resultados fueron revisadas y a veces conversadas para llegar a un acuerdo sobre la mejor decisión. Durante las revisiones, fueron pocas las diferencias entre asignaciones de un revisador con las del otro.

## Revisión de los datos preliminares

Los datos preliminares consisten de 1374 comentarios javadoc formando un total de 8876 frases. Los comentarios javadoc usados consisten cada uno de un bloque de comentario multilinea 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 extraidos de la documentación oficial de Eclipse JFace y Apache Commons abarcando 151 clases de JFace y 17 clases de Apache Commons.[[2]](#footnote-2)

La intención inicial de revisar estos datos era obtener frases de clase no-directivas, juntarlas con las directivas del estudio de Monperrus para obtener un set de datos balanceado – con 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. Pero las revisiones toman tiempo y no fue suficiente como para revisar los 6000 comentarios en total.

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 [[Referencia a sección de resultados pendiente]].

### Método de Revisión de Datos Preliminares

Como se mencionó antes, la clasificación manual de comentarios en 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 a asociado a apoyar el trabajo, Dr. Romain Robbes[[3]](#footnote-3) 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 para evitar errores o bugs, o si es 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

Si no hay más frases, se procede a unir los resultados. Presencialmente los revisadores recorren cada frase y:

* Si no hay diferencias se sigue a la siguiente frase
* De otro modo, se conversa y se llega a un acuerdo, dejando una sola de las clases asignada a la frase

### Observaciones sobre los datos preliminares

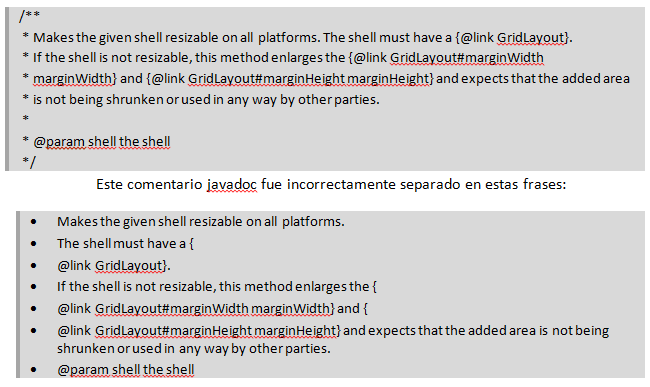
Como se mencionó antes, los datos preliminares fueron usados en experimentos de Text Mining para conseguir información temprana de los resultados. Pero se observaron algunos problemas relacionados con el conjunto de datos.

De partida 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 solo los datos de Jface es un estudio no representativo o menos objetivo. Lo deseable es usar las documentaciones 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.

Los comentarios fueron revisados en orden por paquetes y clases. Esto empeoró aún más la distribución justa de los comentarios, porque como no se revisaron todas las clases de JFace ni de Apache Commons, solo se abarcan algunas clases de estas librerías. Por esto los datos preliminares son demasiado específicos.

Un inconveniente de usar revisar los comentarios en orden, es que se encuentran muchas frases repetidas e idénticas. Comentarios de la misma clase o paquete suelen tener comentarios o partes de ellos repetidos. Incluso hay ciertos comentarios de API que se repiten globalmente en toda una librería. Esto puede causar overfitting en los modelos de predicción de Machine Learning. Overfitting es en Data Mining cuando los datos usados como entrada son muy específicos de cierto dominio y funcionan muy bien para predecir correctamente datos de el mismo dominio, pero funcionan mal prediciendo datos normales que suelen distribuirse por todos los dominios posibles del asunto. En este caso, si solo se usan algunas clases de JFace, se puede obtener un programa que adivine bien directivas en otras clases de JFace, pero mal en documentación de Java, Apache Commons u otras librerías. Incluso puede suceder que adivine mal otras clases de JFace que sean muy distintas a las consideradas por los datos preliminares. Overfitting es algo que se siempre se intenta evitar en Data Mining. En los datos finales se trabaja con un subconjunto aleatorio de cada API para reducir la posibilidad de tener overfitting.

Por otra parte los datos preliminares tienen algunas frases mal separadas. 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. El siguiente es un ejemplo de esto:



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

## Revisión de los datos finales

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

En los datos finales no se incluyen comentarios de campos ni de clases; sólo comentarios de métodos. Se notó que los comentarios de campos y variables suelen ser *no-directivas* formadas por solo una o dos frases, y que los comentarios de clases suelen ser complicados y bastante extensos. Además, clasificar frases de un comentario de clase puede ser complicado porque a veces es necesario una comprensión profunda de esta clase y su relación 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.

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 ya sabe que debe buscar las líneas con *@throws* o *@exception* para ver las Excepciones podría generar un método. No están incluidas en los datos finales para disminuir el ruido de datos no constructivos en el proceso de Text Mining.

Además, todo javadoc con dos o menos frases fue omitido. Estos suelen ser no-directivas de métodos simples que son usualmente son auto-explicativos.

### Método de revisión de los datos finales

A diferencia de los datos preliminares, estos datos fueron manualmente revisados no por ambos revisadores, sinó que por solo uno de ellos. 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 un consejo o una instrucción que no necesariamente es una 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. Por otra parte, las frases asignadas a directiva-null son comentarios que tratan sobre el uso del valor especial *null.* Estos comentarios suelen advertir que algún parámetro puede o no puede ser null, o que cierto método puede retornar null. La regla a seguir es: “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*).

# Aplicación de Text Mining

* Mencionar cuales son los que pueden ser usados como machine learning , aprendiendo con cada set de datos nuevos, usando la api de java.
* Mencionar overfitting y lo que se hizo para evitar tener overfitting

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 [[\*\*\*]] 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 0, 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.

## 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. Las imágenes [[\*\*]] [[\*\*]] y [[\*\*]] muestran la interfaz gráfica de Weka.

[[img de Weka welcome Windows]]

[[img de Weka Explorer]]

[[img de Weka Experimenter]]

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. En este trabajo cada instancia es un vector donde es texto y es uno de los valores {*non-directive,directive,semi-directive,null-directive*} o posiblemente solo {*non-directive,directive*}.

### 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 5.1 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 [[\*\*]].

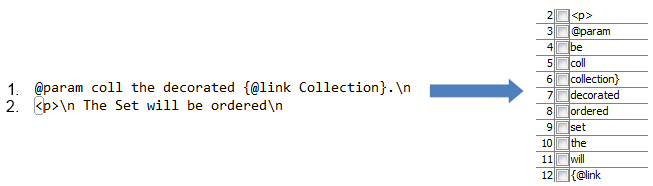


Imagen 5.1: Filtro StringToWordVector siendo aplicado sobre dos instancias de texto resultando en una lista de las palabras que lo forman.

|  |  |
| --- | --- |
| 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 equivalente 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 serparar la frase en sus palabras. Puede ser elegido un ‘*n-gram tokenizer*’ que permite separar la frase en grupos de *n* palabras contiguas. |
| useStoplist | *True es aplicar stopwords* |
| wordsToKeep | Tamaño aporximado del vector resultante |

Tabla 5.1: Parámetros del filtro StringToWordVector probados en los experimentos realizados.

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.

[[\*ver después si es necesario hablar de las pestañas ‘Select Attributes’ y ‘Cluster’\*]]

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

Los clasificadores usados fueron seleccionados por diversas razones. Algunos fueron probados porque son comunes en Data Mining en general, otros son comunes particularmente en Text Mining y otros son mencionados en otros estudios con distintos datos. La siguiente tabla resume a cada clasificador.

|  |  |
| --- | --- |
| Clasificador | Descripción |
| ZeroR |  |
| OneR |  |
| PART | [[ todos pendientes ]] |
| NaiveBayes |  |
| NaiveBayesMultinomial |  |
| BayesianLogisticRegresion |  |
| AdaBoostM1 |  |
| DMNBtext |  |
| LibSVM |  |
| SMO |  |
| RandomForest |  |
| J48 |  |

[[ ToDo: discusión de los parámetros probados ]]

## 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 5.2 y luego se detallan las métricas usadas en este trabajo para evaluar la calidad de los modelos de predicción.

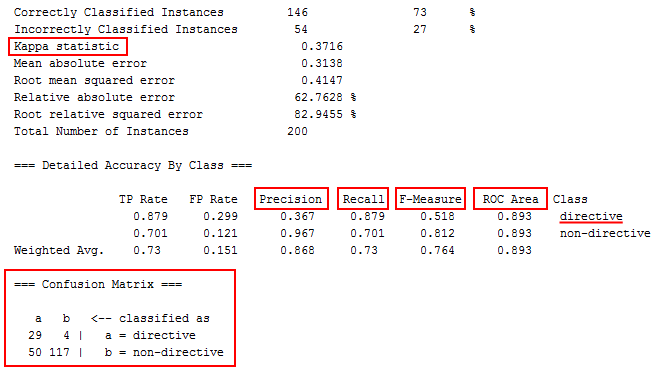


Imagen 5.2: Métricas usadas para evaluar el rendimiento de los clasificadores en este trabajo aparecen destacadas dentro de cuadros en estos resultados de Weka.

Métricas principales:

***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

y como ejemplo, en la Imagen 5.2 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:

Un clasificador con valor de *recall* 0.5 en la clase *directiva* debería adivinar 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 <NombrePrograma> 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 (sección [[\*\*ver caso de uso 2 y (opcionalmente tmbn los: )objetivos\*\*]])

***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:

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 <NombrePrograma> 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.*

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 .

***Kappa statistic***:

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

En este trabajo se consideran los resultados buenos, regulares o malos según la siguiente escala:

|  |  |  |
| --- | --- | --- |
| *F-Measure, Recall, Precision, ROC Area* | Kappa statistic | Calidad del resultado |
| 0.85 – 1 | 0.6 – 1 | Bueno |
| 0.7 – 0.85 | 0.4 – 0.6 | Regular |
| 0.5 – 0.7 | 0 – 0.4 | Malo |

Lo ideal es un clasificador con *F-measure* igual a 1, junto con que ROC Area y Kappa Statistic sean valores buenos según la tabla de arriba. Sin embargo esto es altamente improbable en la práctica es por eso que para comparar modelos se usara el criterio descrito a continuación.

|  |
| --- |
| Criterio de comparación entre modelos de clasificación en Weka |
| Para obtener los mejores clasificadores se compararán los valores de *F-measure*, *recall* y *precision* revisando que los valores de las métricas secundarias sean regular o buenas, y si es necesario analizar con mayor detalle se recurrirá a observar la matriz de confusión de cada clasificador. |

Métricas no usadas:

Otros valores no fueron usados porque no representan fielmente a la calidad de un clasificador para detectar directivas. La Imagen 5.3 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.

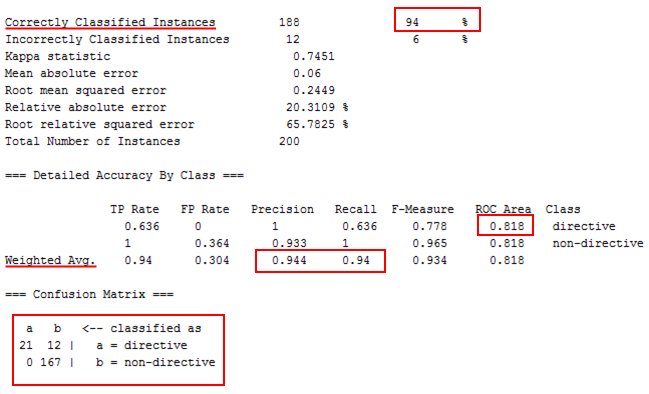


Imagen 5.3: Resultados del clasificador llamado *One-R* sobre los datos preliminares (desbalanceados) para servir de ejemplo de que hay métricas que no representan la calidad de un clasificador para detectar directivas.

Resultados con 4 clases:

Se realizaron experimentos con 4 clases para conocer el rendimiento de los clasificadores intentando clasificar más de solo dos clases. Con cuatro clases de comentarios es más complicado definir una metodología para comparar los resultados como se hizo para los casos con dos clases, basta ver en la Imagen 5.4 que ahora la matriz de confusión es de 4 filas por 4 columnas.

Idealmente se busca que los errores adivinando directivas, semi-directivas y directivas-null sean lo menor posible y al mismo tiempo que pocas *no-directivas* sean incorrectamente adivinadas. En otras palabras: “que las clases *directiva, semi-*directiva y directiva*-null* tengan un valor de *‘recall’* y de ‘*precision’* lo más cercano a 1 posible, mientras que el valor ‘*recall’ de* la clase *no-directiva* sea suficientemente cercano a 1”.

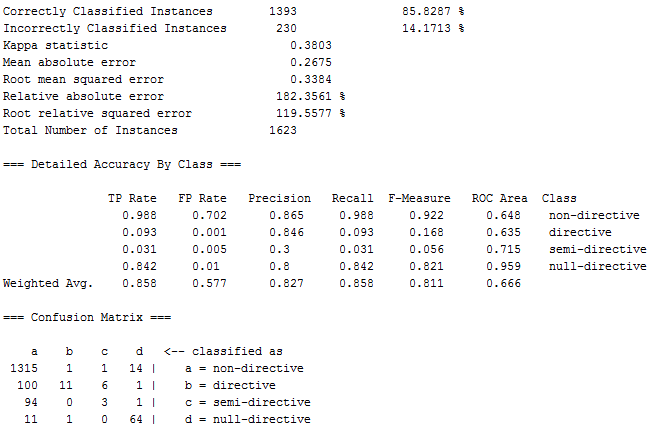


Imagen 5.4: Resultados de Weka como ejemplo de los experimentos realizados con 4 clases.

Sin embargo, los resultados de este trabajo están lejos de alcanzar el ideal buscado y fue necesario comparar los resultados de otra manera. Al menos interesa saber cuánta diferencia existe entre intentar clasificar sólo dos clases (*no-directiva* y *directiva*) versus clasificar cuatro clases. Notemos que las clases *semi-directiva* y *directiva-null* son subclases de *directiva* y que los experimentos con cuatro clases son para obtener información acerca de si es posible adivinar no sólo directivas de no-directivas, sinó que también los subtipos de directivas. Sería mucho mejor poder predecir correctamente los distintos tipos de advertencias en documentación de API para poder destacarlas con diferentes colores por ejemplo, pero si no es realmente factible lograr esto, sería suficiente con poder separar en dos clases. Para comparar entre los resultados con cuatro clases y los con solo dos clases se decide realizar los siguientes dos pasos:

1. Transformar la matriz de confusión a una matriz de 2x2 de esta forma:

1. Comparar *recall* y *precisión* del mismo modo que con resultados con dos clases

Aun teniendo este método, puede ser necesario analizar la matriz de confusión con cuatro clases en detalle para comprender los resultados de cada experimento con cuatro clases.

# 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. Fue usado Java versión 1.8.0\_51 configurado con una memoria máxima de 989.9MB en tiempo de ejecución.

## Resultados Preliminares

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

|  |  |
| --- | --- |
| 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 clasificadores descritos en sección 5.2. |
| StringToWordVector | Parámetros por defectos excepto por:   * *attributeIndices=first* * *lowerCaseTokens=true* |

La siguiente tabla muestra los rendimientos en Weka de los clasificadores usados usando los datos preliminares y ordenados de menor a mayor primero según F-Measure y luego por Recall.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 |
| NaiveBayes | 0.64 | 0.82 | 0.53 | 0.90 | 0.55 |
| NaiveBayesMultinomial | 0.65 | 0.64 | 0.66 | 0.87 | 0.58 |
| BayesianLogisticRegresion | 0.70 | 0.67 | 0.73 | 0.81 | 0.06 |
| PART | 0.76 | 0.76 | 0.76 | 0.89 | 0.71 |
| RandomForest | 0.76 | 0.67 | 0.88 | 0.97 | 0.72 |
| OneR | 0.78 | 0.64 | 1 | 0.82 | 0.75 |
| 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 en general hay mejores resultados con LibSVM, SMO y J48 clasificando directivas sobre un set de datos desbalanceado, en el buen sentido pues los comentarios de API en la realidad suelen tener un desbalance entre directivas y no-directivas porque hay menos directivas que comentarios normales.

Sin embargo, si se quiere encontrar a los clasificadores que mejor detectan a la mayor cantidad de directivas posibles, los resultados se deberían ordenar por el valor de *recall*. Esto resulta en que los mejores para este propósito son nuevamente J48, SMO y LibSVM, pero notemos que Naive Bayes sube 4to lugar, PART sube a 5to lugar y por otra parte OneR baja hasta el 8vo lugar.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 |

Por otra parte, si se quiere encontrar a los clasificadores que menos ensucian al grupo de frases destacadas con comentarios normales que son confundidas por directivas, los resultados se deberían ordenar por el valor de *precision*. Esto resulta en que los mejores para este propósito son OneR, RandomForest y DMNBtext. Notemos que NaiveBayes queda en penúltimo lugar.

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

## Resultados Finales

A continuación se presentan varias combinaciones entre sets de entrenamientos y sets de pruebas 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 *directives* buscando valores altos en *recall* y *F-Measure*. El otro objetivo de varios experimentos es descubrir con qué porcentaje de los datos al entrenar entrenamiento es suficiente para lograr resultados aceptables.

Los experimentos fueron 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. |
| 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) con varios clasificadores. |
| StringToWordVector | Parámetros por defectos 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. |

### 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 3.1.1y en 3.2.1, significando que hay que optimizar el valor de *recall*. Pero los resultados de un experimento mostrados en la Tabla 6.1y en la Imagen 6.1 muestra optimizar el F-Measure puede lograr resultados con menos *falsos negativos*. El *recall* más alto es 0.93 de *NaiveBayes*, pero al mismo tiempo tiene bajos valores de *precision* y *F-Measure*. *RandomForest* y *SMO* tienen un valor de *F-Measure* alto pero no alcanza a *NaiveBayes* hablando de *recall*.

Usando una *matriz de costos* en Weka se puede incentivar a que un clasificador mejore en *recall* pero empeore en *precision*, o viceversa. Usando una matriz de sobre *SMO* y una matriz de sobre *RandomForest* se obtienen las matrices de confusión de la logrando el mismo *recall* (o mejor) que *NaiveBayes*, pero mejorando el valor de *precision*.

Tabla 6.1: Resultados entrenando con datos finales de Apache Commons, Java y el primer 10% de JFace; set de prueba, 90% restantes de JFace.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | F-Measure | recall | precision | ROC Area | kappa |
| ZeroR | 0.00 | 0.00 | 0.00 | 0.50 | 0.00 |
| OneR | 0.39 | 0.29 | 0.57 | 0.63 | 0.33 |
| Logistic | 0.39 | 0.66 | 0.28 | 0.70 | 0.26 |
| IBk | 0.53 | 0.45 | 0.65 | 0.79 | 0.47 |
| NaiveBayes | 0.54 | **0.93** | 0.38 | 0.91 | 0.43 |
| NaiveBayesMultinomial | 0.56 | 0.87 | 0.41 | 0.90 | 0.46 |
| BayesianLogisticRegression | 0.59 | 0.80 | 0.46 | 0.83 | 0.50 |
| AdaBoostM1 | 0.59 | 0.81 | 0.47 | 0.90 | 0.51 |
| J48 | 0.60 | 0.79 | 0.48 | 0.90 | 0.52 |
| PART | 0.60 | 0.83 | 0.47 | 0.90 | 0.52 |
| LibSVM | 0.66 | 0.87 | 0.53 | 0.88 | 0.59 |
| SMO | **0.66** | 0.87 | 0.53 | 0.88 | 0.59 |
| DMNBtext | 0.66 | 0.87 | 0.54 | 0.94 | 0.60 |
| RandomForest | **0.71** | 0.77 | 0.67 | 0.96 | 0.67 |

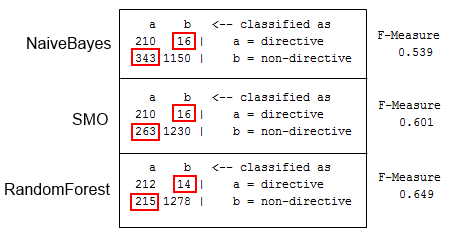


Imagen 6.1 Matrices de confusión mostrando que SMO y RandomForest superan en este caso a NaiveBayes porque tienen ambos un valor de F-Measure significativamente mayor a este y pueden usar una matriz de costos para mejorar sus valores de recall.

Sin embargo, 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* entre la y la Tabla 6.1. Esto indica que ambos clasificadores pueden superar a *NaiveBayes* mientras su valor de *F-Measure* finales no resulten inferiores al *F-Measure* de *NaiveBayes*.

Otro 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 ve en la sección 6.2.6.

### 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 6.2: 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 6.3: 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 6.3 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.

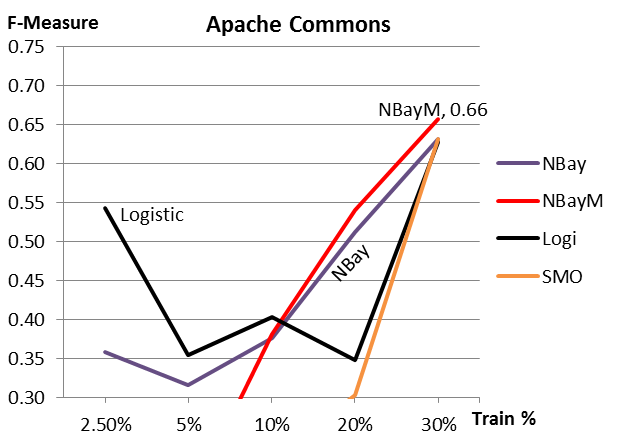
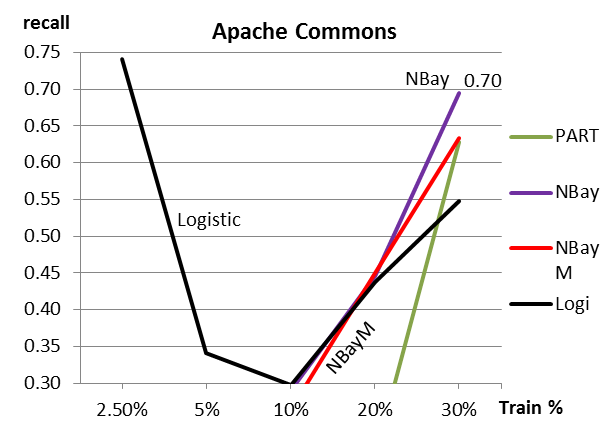
Posiblemente la estrategia de keywords puede ser mejorada para obtener mejores resultados. Sin embargo tiene la desventaja que no puede ser usada con la matriz de costos para acercar más el valor de *recall* hacia 1, al menos en su algoritmo básico usado en este trabajo.

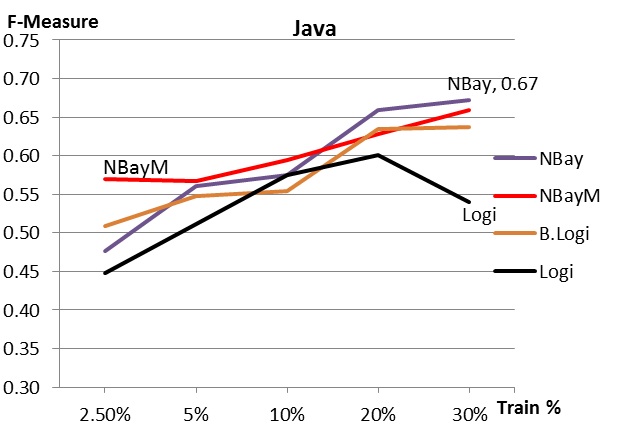
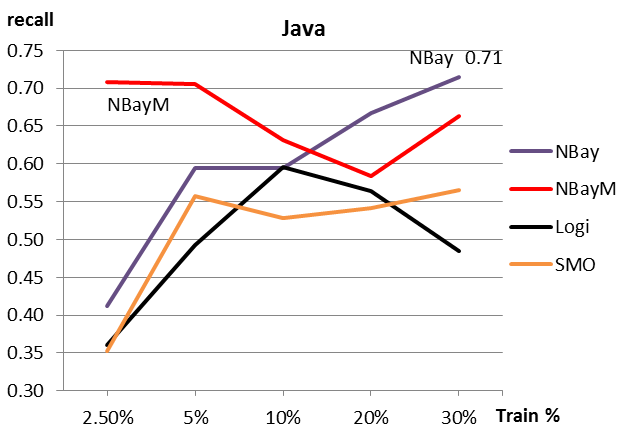
### Usando una API

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. 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 y sin tener ningún modelo previo de cómo detectar las directivas. Esto haría posible clasificar manualmente muy pocas directivas de una documentación y detectar el resto automáticamente.

Los resultados se encuentran graficados en la Imagen 6.2.Al entrenar con un 2.5% que son entre 40 a 60 instancias o entrenar con un 5% que son entre 80 a 120 instancias, ocurrió que Logistic en dos casos y NaiveBayesMultinomial en un caso obtuvieron un *recall* alto y al mismo tiempo un *F-Value* aceptable. 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. En Apache Commons y JFace funcionó bien el clasificador *Logistic*, pero no fue así en los datos de Java. En los tres casos, cuando un clasificador funciona bien en 2.5%, empeora a medida que el porcentaje de entrenamiento aumenta hacia 30%.

En general los resultados no son muy prometedores. Sólo en los datos de JFace se logró obtener un *recall* mayor a 8.5 y un *F-Measure* cercano a 6.0. Los resultados apoyan a que para detectar directivas puede no ser suficiente entrenar con un 30% de esa API y detectar el resto





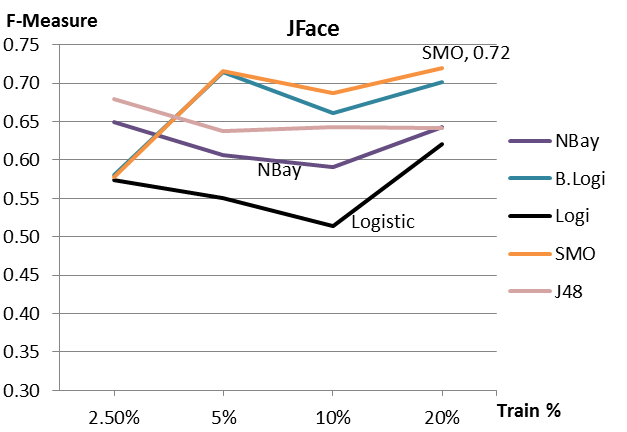
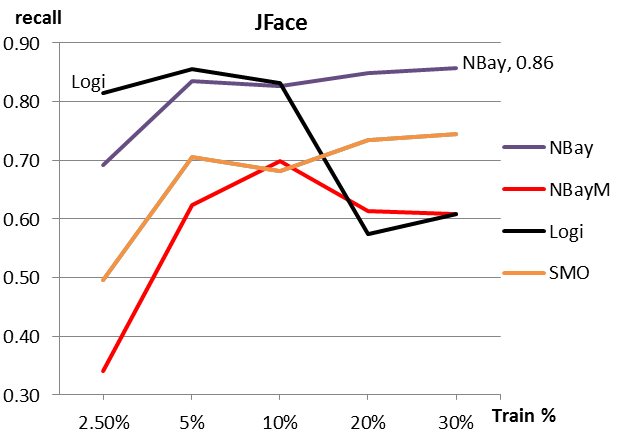


Imagen 6.2: Entrenando con un porcentaje de cada API y usando el resto de los datos como set de pruebas.

### 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 y están ordenadas en orden creciente según los valores de *recall* y los mejores valores aparecen destacados.

Tabla 6.4: 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 6.5: 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 6.6: 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)* | | | |

Los resultados son regulares a excepción del caso 2 entrenando con Java el cual es un buen resultado. Sin embargo se observan mejores resultados entrenando con dos librerías y realizando pruebas en la API restante, como se muestra en la sección siguiente. Pero por otra parte al realizar pruebas con más instancias la cantidad de errores suele aumentar haciendo algo injusta la comparación entre ambos experimentos.

### Entrenando con 2 API y probando con 1 API

Imagen 6.3: Graficos 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.

Los resultados de la Imagen 6.3 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. 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. Además una matriz de costos podría aplicarse a NaiveBayes y a SMO para mejorar aún más el *recall*. Los clasificadores con peor rendimiento no fueron incluidos para mejorar la claridad de los gráficos.

Imagen 6.4: Graficos 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 6.4 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 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.

Imagen 6.5: 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 – Java y JFace con un porcentaje de Apache Commons – se alcanzaron mejores resultados nuevamente con NaiveBayesMultinomial, solo que esta vez presenta una caída de F-Measure a los 50%. Esta vez NaiveBayes no comienza muy bien en 0% pero mejora bastante cuando alcanza 50%. Los resultados se pueden visualizar en en la Imagen 6.5.

Estos experimentos apoyan a que los mejores clasificadores son NaiveBayes y NaiveBayesMultinomial cuando para adivinar ya se tiene en un modelo el conocimiento por aprendizaje de máquina de dos librerías. Efectivamente en el primer experimento el mejor parece ser SMO porque se le puede aplicar una matriz de costos para alcanzar el valor de 0.94 en *recall* que tiene NaiveBayes en 50%, pero al hacer esto probablemente bajaría el valor de *precision* de SMO hasta un valor cercano al *precision* de NaiveBayes. Además SMO sólo presenta un buen rendimiento en el primer set de datos.

### 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 6.7 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 6.7: 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)* | | | | | | | | | | | | | |

Usándo 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 – ‘Usando una API’. Los resultados de *Logistic* entrando sólo con un 2.5% de una API son buenos, pero no consistentes. Pueden ser un accidente que haya funcionado bien o también puede ser que sea un buen clasificador con porcentajes muy bajos de entrenamiento. Sin embargo los costos de matriz solo empeoró el rendimiento de *Logistic* como lo muestra la Imagen 6.6.

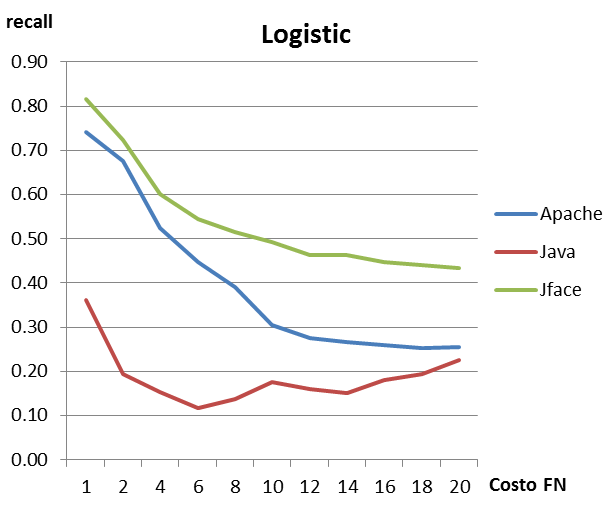


Imagen 6.6: Efecto de matriz de costos sobre *Logistic*   
entrenando con un 2.5% de cada API y realizando   
pruebas con el porcentaje restante.

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

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[[4]](#footnote-4) y el otro es una modificación de ese mismo acorde al 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.

La Tabla 6.8 muestra que los resultados son muy similares por lo que se decide usar la lista de Google sin modificaciones. Las combinaciones con wordcounts fueron hechas pensando que el no usarlas podría significar un mal rendimiento para NaiveBayesMultinomial el cual está diseñado para recibir la frecuencia de cada palabra en una frase.

Tabla 6.8: Comparación de rendimiento al usar dos listas de stopwords.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | NaiveBayes | | NaiveBayesMultinomial | |
| Stopwords | Wordcounts | recall | f-measure | recall | f-measure |
| **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 mismos clasificadores para conocer cuál permite obtener mejores resultados. En la Tabla 6.9 se muestra que los resultados son muy similares pero favoreciendo por poco a *Snowball Stemmer*, por lo que se decide usarlo para los experimentos posteriores.

Tabla 6.9: Comparación de rendimiento al usar distintos stemmers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 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*. Las transformaciones TF-IDF ayudan a considerar el peso de las palabras comunes versus las palabras infrecuentes y a veces pueden ayudar en problemas de Text Mining.

Los resultados de las 16 combinaciones de cada API se encuentran el Anexo - 12.1. 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.

### Tiempos de entrenamiento y de prueba

La Imagen 6.7 muestra el tiempo en segundos que tarda cada clasificador estudiado sobre 2951 instancias. Esta información puede ser útil al momento de elegir entre dos clasificadores con rendimientos muy similares, eligiendo por supuesto al más rápido.



Imagen 6.7: Tiempos de entrenamiento y prueba de cada clasificador sobre  
2951 instancias al entrenar y 2952 instancias en las pruebas

# Discusión general

* Mencionar en alguna parte lo de sublime text (y quizás eMoose)

# Limitaciones y problemas de validez (?)

* Datos finales no incluyen comentarios de clases. Esto podría perjudicar el rendimiento los clasificadores entrenados al encontrar un comentario de clase.
* Datos finales mejor diseñados, pero solo revisados por una persona.

# Conclusiones y trabajo futuro

1 – probar con directive, non-directive + la clase null-directive

# Glosario

# Bibliografía

x

|  |  |
| --- | --- |
| [1] | J.D. Herbsleb U. Dekel, "Reading the Documentation of Invoked API Functions in Program Comprehension," ICPC, 2009. |
| [2] | R. Lampert, S. Letovsky, D. Littman, and J. Pinto E. Soloway, "Designing documentation to compensate for delocalized plans," ACM, 31(11):1259–1267, 1988. |
| [3] | B. Hummel, E. Juergens D. Steidl, "Quality Analysis of Source Code Comments," 2013. |

x

# Anexo

## Stop-words, stemming, TF-IDF Transform y wordcounts

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 | Wordcounts | 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 | Wordcounts | 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 | Wordcounts | 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)* | | | | | | | | | | | | |

1. Para el detalle sobre el clasificador de Machine Learning ver [[\*ref a modelo de ML usado en API de Weka\*]] [↑](#footnote-ref-1)
2. Lista con todas las clases abarcadas puede ser encontrada en <https://github.com/gabocorrea/Datos-Publicos-del-Trabajo-de-Titulo/> [↑](#footnote-ref-2)
3. <http://users.dcc.uchile.cl/~rrobbes/> [↑](#footnote-ref-3)
4. Google stopwords en https://code.google.com/p/stop-words/Ñ [↑](#footnote-ref-4)