

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](#_Toc427070458)

[1.1 Documentación de API 1](#_Toc427070459)

[1.2 Problema y Motivación 1](#_Toc427070460)

[1.3 Una Solución 1](#_Toc427070461)

[1.4 Objetivos 1](#_Toc427070462)

[1.4.1 Objetivos Generales 2](#_Toc427070463)

[1.4.2 Objetivos Específicos 2](#_Toc427070464)

[1.4.3 Metodología 2](#_Toc427070465)

[2 Marco teórico y trabajos anteriores 2](#_Toc427070466)

[2.1 Importancia de comentarios en API y código fuente 2](#_Toc427070467)

[2.2 Destacación de directivas en API 2](#_Toc427070468)

[2.2.1 Problema: Las directivas pueden pasar desapercibidas 2](#_Toc427070469)

[2.2.2 Syntax highlighting en editores de texto 2](#_Toc427070470)

[2.2.3 eMoose y tagSea 2](#_Toc427070471)

[2.3 Trabajo de Monperrus et al. 2](#_Toc427070472)

[2.3.1 Keywords para detectar directivas 2](#_Toc427070473)

[2.3.2 Taxonomía de directivas 2](#_Toc427070474)

[3 Desarrollo de <NombrePrograma> 2](#_Toc427070475)

[3.1 Objetivo General de la Herramienta 2](#_Toc427070476)

[3.2 Diseño de <NombrePrograma> 4](#_Toc427070477)

[3.3 Detalles de la Implementación 5](#_Toc427070478)

[3.4 Trabajo Futuro para <NombrePrograma> 6](#_Toc427070479)

[4 Revisión Manual de Comentarios de API 6](#_Toc427070480)

[4.1 Revisión de los datos preliminares 7](#_Toc427070481)

[4.1.1 Metodo de Revisión de Datos Preliminares 7](#_Toc427070482)

[4.1.2 Observaciones sobre los datos preliminares 8](#_Toc427070483)

[4.2 Revisión de los datos finales 9](#_Toc427070484)

[5 Aplicación de Text Mining 11](#_Toc427070485)

[6 Resultados 19](#_Toc427070486)

[6.1 Resultados Preliminares 19](#_Toc427070487)

[6.2 Resultados Finales 21](#_Toc427070488)

[7 Discusión general 21](#_Toc427070489)

[8 Limitaciones y problemas de validez (?) 21](#_Toc427070490)

[9 Conclusiones y trabajo futuro 22](#_Toc427070491)

[10 Glosario 22](#_Toc427070492)

[11 Bibliografía 22](#_Toc427070493)

[12 Anexo 22](#_Toc427070494)

# Introducción

## Documentación de API

Explicar que es la documentación de api y más.

## Problema y Motivación

Directivas no se ven. Esto hace mas probable la existencia de bugs. Intentar escribir mejor documentación suena como una solución, pero puede haber varios problemas para que funcione en la práctica [[falta una referencia a algún estudio para la ultima frase]]

Una directiva es … [[definición corta aca]]. .. [[Para una explicación detallada del concepto ver [°n] o Anexo/Directivas]]

## Una Solución

[[Hablar de destacar directivas y de la necesidad de mayor automatización pues la asignación manual de directivas es muy tedioso y toma mucho tiempo.]]

Una solución – la propuesta por este trabajo – es que se detecten de manera semiautomática las directivas de API usando las herramientas de Text Mining y Machine Learning, usando comentarios reales de librerías API como base y aprendizaje.

Es posible que un programa aprenda a detectar directivas, la duda es ¿con cuánta eficacia o precisión lo hace? El rendimiento de este método es medido en el presente trabajo en una serie de experimentos usando Text Mining, y es comparado con el rendimiento de otro método: si un comentario contiene alguna palabras clave, o *keyword*, de cierto conjunto preestablecido, entonces es una directiva.

La solución incluye un software que facilita el pre-procesamiento de los comentarios de API necesario para la aplicación de Text Mining. Permite realizar una asignación de directiva/no-directiva (y posiblemente más tipos) de cada phrase de los comentarios de una API y luego entregar los resultados en un formato compatible con la aplicación de Text Mining.

## Objetivos

(parte similar al informe de CC6908)

### Objetivos Generales

### Objetivos Específicos

### Metodología

# 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

### 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.

## Objetivo General, Usuario Final y Casos de Uso

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.

Los usuarios finales son principalmente dos. A continuación se describen ambos usuarios a los cuales está destinada la aplicación, y un caso de uso de ejemplo por cada uno.

**Persona a cargo de la Documentación de una API**

Es una persona interesada en 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 usa el programa para que las directivas sean detectadas y destacadas en su documentación automáticamente. Luego si desea puede revisar y corregir manualmente las directivas que fueron detectadas usando la aplicación.

Caso de Uso #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 contiene otros archivos .txt y .html, pero estos no son considerados. Para terminar de crear el proyecto solo debe elegir si quiere que el programa intente adivinar por si sólo cuales son las directivas o si no quiere que el programa sugiera cuales son las directivas. 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 tiene la aplicación.

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

El usuario decide revisar (y corregir si es necesario) algunas frases de la API en la página web, usando las flechas y tecla de espacio de su teclado, y cuando ha terminado exporta el proyecto a una corregida documentación de API que deja marcadas las frases que son directivas[[2]](#footnote-2).

**Investigador de Text Mining sobre Comentarios de Código**

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.

Caso de Uso #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.

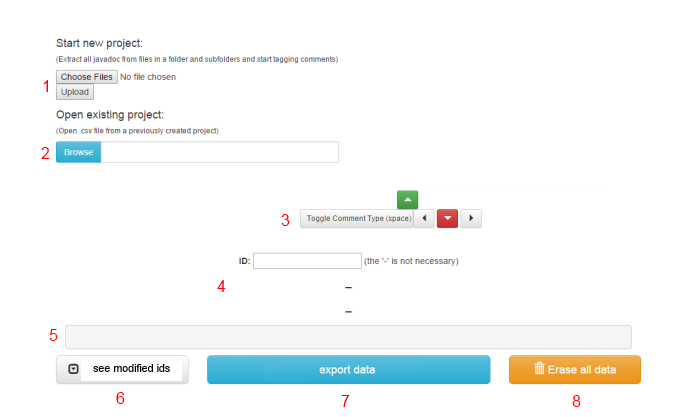
Luego de seleccionar la carpeta, la página toma unos segundos en procesar los archivos y luego muestra el primer comentario de API, indicando que el proyecto ya ha sido creado y está listo para ser manualmente revisado.

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.

## 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

* Explicar Datos del trabajo de Monperrus.
* Preparación de los datos de Monperrus, venían en .xml. los necesitamos en .csv

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*.

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 1894 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.[[3]](#footnote-3)

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[[4]](#footnote-4) 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

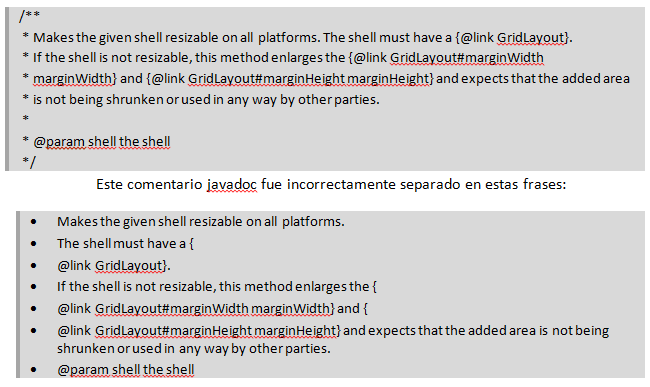
Como se mencionó antes, los datos preliminares fueron usados en experimentos de Text Mining para conseguir información temprana de los resultados. Pero se observaron algunos problemas relacionados con el conjunto de datos.

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 1500 comentarios javadoc, conteniendo [[<numero-frases>]] frases, extraidos de las documentaciones de API de JFace, Apache Commons y Java. A diferencia de los datos preliminares, incluyen las tres API tomando 500 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 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.

## 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 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 [[\*\*]].

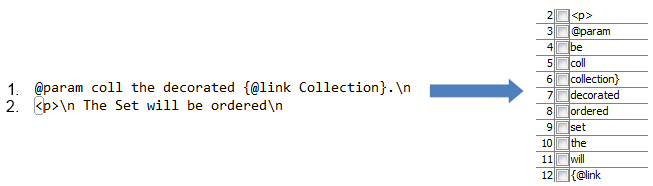


Imagen : Filtro StringToWordVector siendo aplicado sobre dos instancias de texto resultando en una lista de las palabras que lo forman.

|  |  |
| --- | --- |
| 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 : 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 5.3.

[[\*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  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 1 y luego se detallan las métricas usadas en este trabajo para evaluar la calidad de los modelos de predicción.

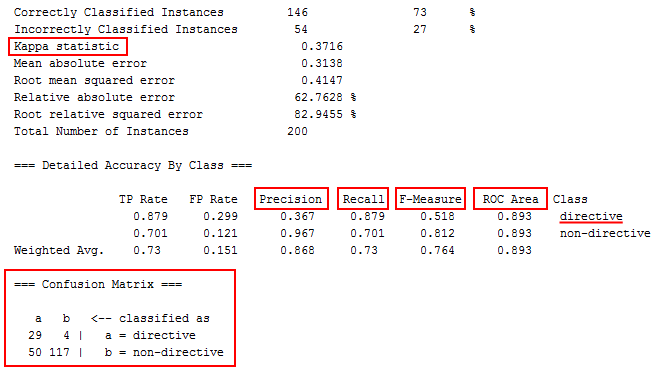


Imagen : Métricas usadas para evaluar el rendimiento de los clasificadores en este trabajo aparecen destacadas dentro de cuadros en estos resultados de Weka.

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 1 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 a el valor 1, mientras que un valor de 0.5 es comparable a adivinar las clases por azar.

***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 2 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.

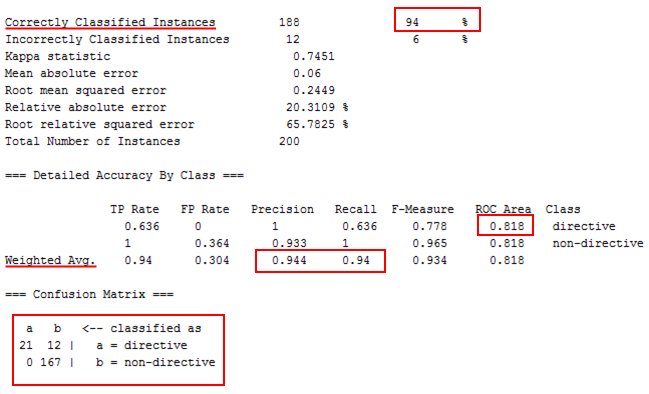


Imagen : Resultados del clasificador llamado *One-R* sobre los datos preliminares (desbalanceados) para servir de ejemplo de que hay métricas que no representan la calidad de un clasificador para detectar directivas.

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 3 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”.

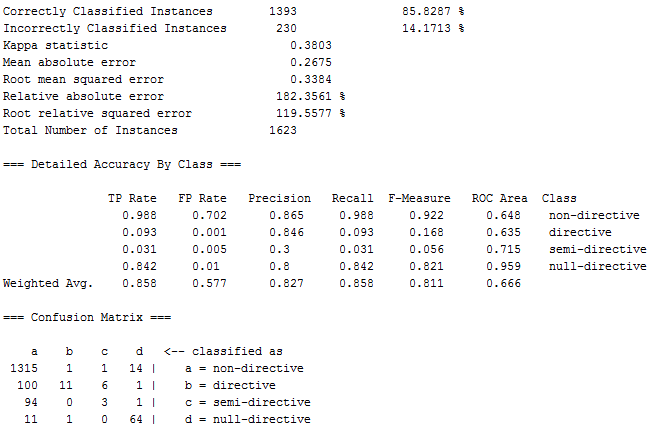


Imagen : Resultados de Weka como ejemplo de los experimentos realizados con 4 clases.

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.

## Discusión

[[.. pendiente]]

# 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 5.3: ‘Evaluación del rendimiento de un clasificador’.

## Resultados Preliminares

* Tablas comparando distintos clasificadores, usando o no stemming, usando o no stopwords
* Tablas adicionales comparando otras opciones y parámetros del filtro y opciones del mejor clasificador (puede que no sea tan relevante incluir esto)
* Set reducido de clasificadores para los resultados finales

Los resultados preliminares provienen de los experimentos realizados en 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 |

[[… me falta poner los resultados con el set de datos *balanceado* ..lo anterior es con los datos balanceados]]

[[.. me falta poner los resultados del comparador con el Experimenter de Weka]]

[[.. me falta agregar los resultados de las comparaciones con stemmin, stopwords entre los 3 o 4 mejores clasificadores]]

[[..me falta agregar al anexo las fotos de todos los resultados preliminares]]

## Resultados Finales

* Hablar de los resultados con el set de datos final (incluir una referencia a cuales son los datos finales, para que no haya confusión) Discusión de los resultados
* Tablas comparando distintos clasificadores, usando o no stemming, usando o no stopwords
* Tablas adicionales comparando otras opciones y parámetros del filtro y opciones del mejor clasificador
* Comparación con heurística/regla de Keywords de Monperrus u otros estudios

# 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

# Glosario

# Bibliografía

# Anexo

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. La exportación de un proyecto en la página web a una documentación de API corregida no está implementada aun en <NombrePrograma>. Sin embargo, es así como se tenía planeado que debería ser. [↑](#footnote-ref-2)
3. Lista con todas las clases abarcadas puede ser encontrada en <https://github.com/gabocorrea/Datos-Publicos-del-Trabajo-de-Titulo/> [↑](#footnote-ref-3)
4. <http://users.dcc.uchile.cl/~rrobbes/> [↑](#footnote-ref-4)