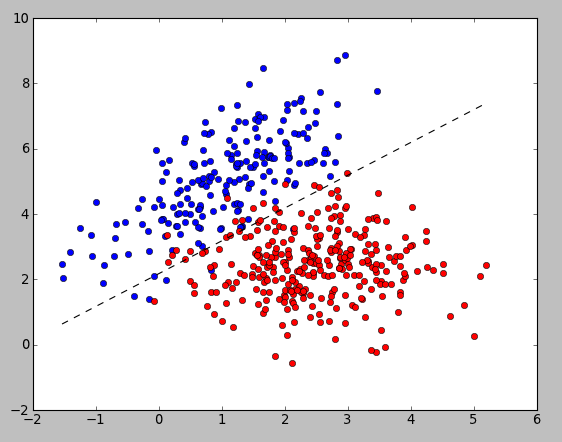
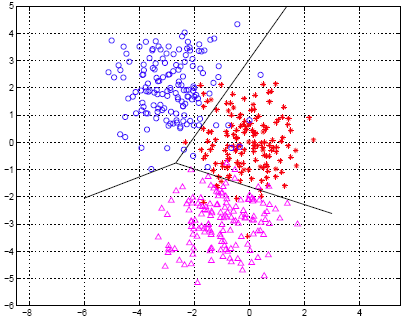
# DATA MINING PRÁCTICA 5





Egileak: Javier Fernández, Ander Orbegozo, Alex Valero, Unai Urkidi, Mikel Arrospide, Xabier Apraiz

## ÍNDICE:

1.- PLANTEAMIENTO DEL PROBLEMA………………………………………………………………………………………3

2.- REPARTO DEL TRABAJO……………………………………………………………………………………………………….4

3.- PROBLEMAS TENIDOS Y SOLUCIONES…………………………………………………………………………………5

4.- EXPERIMENTOS Y PRUEBAS………………………………………………………………………………………………..6

5.- CONCLUSIONES…………………………………………………………………………………………………………..………8

6.- CÓMO INICIAR EL PROGRAMA………………………………………………………………………………………..…8

7.- NOTAS, COMENTARIOS Y ADVERTENCIAS…………………………………………………………………………10

## 1.-PLANTEAMIENTO DEL PROBLEMA

Primeramente, para afrontar esta práctica de trabajo conjunto de la clase, nos dividimos en varios subgrupos (el nuestro de 6 personas), donde cada uno abordaría un apartado del problema.

Así pues, en nuestro grupo decidimos encargarnos de implementar algún filtro (como el “Resample” y/o el “Smote”) para poder hacer frente a los ficheros, inicialmente muy cargados de datos y con un desequilibrio entre los valores positivo y negativo de la clase, y varios sistemas de clasificación (como el “RandomForest”, “Support Vector Machine” o “SVM”, “OneClass”, “NaiveBayes” y “BayesNet”).

Como se puede ver, son muchos los clasificadores que decidimos implementar, por lo que entrenar con ellos para saber cuáles son los valores de los parámetros con los cuales se conseguía un *fmeasure* mejor con respecto al valor de la clase positiva resulta muy trabajoso.

De ahí que decidiésemos implementar, primeramente, un filtro como el “Resample”, que se encarga de reducir el tamaño del dataset inicial (22000 instancias, más o menos; con una diferencia entre clase positiva y negativa de 21980 vs 80), equilibrándolo, a su vez, y consiguiendo que la diferencia entre instancias con estimación de clase positiva y negativa fuese mínima.

En este punto, teníamos la duda de si intentar implementar el “Smote” nos iba a merecer la pena o no, ya que con el “Resample” se podía conseguir un resultado muy similar. Empezamos a implementar ambos módulos, individuales, pero al ver que el filtro del “Smote” nos daba problemas dependiendo de las librerías y las versiones de Java, decidimos centrarnos en el “Resample” y en la implementación y tratamiento de los clasificadores.

Una vez conseguimos el módulo ejecutable del “Resample”, sólo teníamos que integrarlo en el código de los clasificadores (también módulos independientes) para que el entrenamiento con el dataset de entrenamiento original no fuese tan largo, además de ese desequilibrio ya comentado entre los valores de la clase, el cual producía que al entrenar el modelo de los clasificadores el *fmeasure* de la clase positiva fuese muy baja.

Nos llevó trabajo implementar los clasificadores, ya que teníamos que decidir, primero de todo, qué parámetros queríamos tratar para buscar su valor óptimo (de cara a una clasificación cuyo modelo, después del entrenamiento, nos diese un *fmeasure* alto con respecto a la clase positiva), y después, ejecutar las pruebas `para comprobar su validez. Decidimos, pues, sacar los valores óptimos impresos por pantalla, y/o guardarlos en un fichero de texto plano, y extraer, también, el modelo para poder cargarlo, posteriormente en un módulo que se encargaría de hacer las estimaciones para el fichero de test.

Finalmente, para el paso final, decidimos generar un Script que ejecutara los módulos individuales tanto del filtro como de los distintos clasificadores, para que se crearan los modelos de clasificación. Otro script se encargaba de, al igual que el script anterior que cogía los módulos y los ejecutaba, coger los modelos generados recientemente y aplicarlos sobre el fichero de test, para sacar las predicciones respectivas de cada modelo de clasificación. Para terminar, nos valimos de la ayuda de la implementación de un módulo que se encargaba de ver qué predicciones se conseguían con la ejecución del segundo script mencionado, y, elegir, por votación, cuál era el valor de clase más votado en la estimación final para cada instancia del dataset del test, y así, decidir cuál era la clase estimada final.

## 2.-REPARTO DEL TRABAJO

El reparto de trabajo ha ido variando a medida que se iba avanzando en la práctica. En un principio, estos eran las tareas repartidas a cada miembro:

Javier Fernández -> Implementación de filtro “Smote” y de clasificador “SVM”

Unai Urkidi -> Implementación de filtro “Resample”

Alex Valero -> Implementación de clasificador “BayesNet”

Ander Orbegozo -> Implementación de clasificador “OneClassClassifier”

Mikel Arrospide -> Implementación de clasificador “BayesNet”

Xabier Apraiz -> Implementación de clasificador “RandomForest”

Pero, a la vista de que algunos compañeros implementaron algunos módulos más rápido que otros el reparto fue variando, de manera que unos se ayudaban a otros en la consecución de los objetivos. Ander y Alex consiguieron implementar sus módulos individuales inicialmente propuestos, mientras que Unai y Javi, entre ambos, consiguieron implementar el “Resample” y el “SVM”, dejando el filtro “Smote” como objetivo secundario por lo mencionado en el apartado anterior. Mikel y Xabi también consiguieron cumplir sus respectivos objetivos. Al final, todos terminamos haciéndolo todo y ayudándonos con todo (scripts, módulos, documentación, etc.).

Aparte de eso, también generamos un ejecutable en java que servía para contar el número de instancias que había por cada valor de clase en el dataset, con el objetivo de que nos sirviera de ayuda para saber si el “Resample” y/o el “Smote” nos funcionaban como esperábamos. Así, con la ayuda de ese módulo, pudimos conseguir probar diferentes reducciones del tamaño del dataset por medio del “Resample” (distintas especificaciones del porcentaje a reducir), y decidir quedarnos con la reducción de instancias que menos diferencia entre valores de clase generase.

Los scripts corrieron cargo de todo el grupo, aunque fuese Ander el que más contribuyó en el proceso, al igual que la implementación del módulo de decisión de estimación final lo hizo Alex.

## 3.-PROBLEMAS TENIDOS Y SOLUCIONES

Muchos de los problemas que hemos tenido han sido más relativos a cuestiones internas del código Java que al entendimiento de uso de los clasificadores y/o filtros.

**SMOTE:** Como problema destacado podemos mencionar el uso del módulo “Smote” en Java. La implementación inicial que había conseguido hacer Javi había resultado exitoso; tanto que el jar ejecutable funcionaba correcta y debidamente, habiendo tratado los parámetros de porcentaje y *nearestNeighbors* para buscar una generación de instancias nuevas óptima en base al filtro. Pero, parece ser que el problema era que la versión con la que él había trabajado era de un JDK más antiguo que el del resto, y había errores de compilación. Además, a pesar de que ese módulo funcionaba para un JDK antiguo, el resto de los módulos cono podían ser el “OneClass” o el “SVM” no conseguían funcionar en esa versión. De modo que, tras laboriosas jornadas de trabajo sobre el “Smote”, se decidió dejarlo de lado con el objetivo de terminar el resto de la práctica, y a la vista de que el “Resample” podía servir, perfectamente, también para cumplir con la idea de reducción y equilibrio del dataset de entrenamiento.

**BAYESNET:** En la implementación del clasificador BayesNet el único problema ha sido el alto número de atributos e instancias de los ficheros propuestos para el problema. La solución que hemos tomado ha sido aplicar filtros y utilizar otros archivos más simples para comprobar su correcto funcionamiento.

**SAILLKATZE FASEA:** En el módulo “SailkatzeFasea” queríamos cargar las predicciones de cada clasificador en un arff pero nos encontramos con el problema de no poder crear un conjunto de instancias vacio y rellenarlo con las predicciones recibidas de cada modelo. Por tanto, decidimos coger el conjunto de instancias del fichero Test y añadirle otro atributo a cada instancia con la predicción de cada clasificador.

**EMAITZAK KUDEATU:** En el siguiente paso, en el módulo “EmaitzakKudeatu” recibimos como entrada ése fichero arff y le pasamos por parámetro cuántos atributos tiene que tomar como resultados de los clasificadores. Tuvimos que decidir qué criterio íbamos a aplicar para decidir la estimación final de la clase. Consideramos que la votación era la más adecuada y la implementamos sacando los resultados en un fichero de texto.

**ONE CLASS:** La primera librería que utilizamos fue “libSVM”. Esta Liberia nos dio varios problemas. El primer error fue que no conseguía leer el *classpath* a pesar de que lo tenía puesto en las variables de entorno. El error lo arreglamos descargando otra versión de “weka” y “LibSVM” que ya venían preparados y no requerían de la variable *classpath*.

Con archivos con pocos datos funcionaba bien pero al intentar usarlo con archivos más grandes nos saltaba el error de *outofmemory* incluso añadiéndole a la VM de Java más memoria.

Teniendo en cuenta estos problemas buscamos otra librería y encontramos una librería que viene muy bien integrada con “weka”. Esta librería al poder ser utilizada del mismo modo que cualquier clasificador de “wek”a nos ayudó a implementarlo más rápido y compensar el tiempo utilizado en “libSVM”.

**NAIVE BAYES:** A la hora de implementar el clasificador Naive Bayes no hemos tenido ningún problema ni con compatibilidades con las librerías ni con el código y tampoco ha dado ningún error con ningún tipo de archivo, fuera cual fuera el numero de instancias o atributos. Junto al modelo también creamos en el mismos lugar en el que queremos que se crea el modelo un eval.txt en el que se muestra la evaluación del modelo conseguido.

## 4.-EXPERIMENTOS Y PRUEBAS

Lara probar los distintos módulos hemos utilizado varias estrategias.

Para la prueba de los módulos encargados de ejecutar los filtros, hemos utilizado no sólo los ficheros de train y dev referentes a la práctica, sino también otros muchos arff-s, para ver que funcionaba bien.

Para probar los clasificadores y decidir qué valores de parámetros escoger sí que hemos utilizado el fichero de train y el de dev, ya que necesitábamos resultados que nos fuesen representativos para los datos de la práctica.

**RESAMPLE:** Para este filtro, hemos hecho un módulo que trataba el parámetro del porcentaje del dataset con el que nos queríamos quedar. Para ello, hemos hecho una búsqueda exhaustiva comprobando cada 10 unidades porcentuales cuál era el resultado obtenido, aplicando el método *biasToUniform* que equilibraba los valores de la clase, y nos quedamos con el resultado que índice menor diferencia entre el valor positivo y el negativo de la clase. Por ejemplo, si la clase + tiene 8024 instancias y la – tiene 8034, este resultado es mejor que si fuesen 3030 y 3050, pues en el primer caso, aunque haya más instancias totales, la diferencia entre valores positivo y negativo de la clase es menor. Así pues, mediante esos dos parámetros hemos conseguido implementar este módulo que devuelve un fichero en formato arff con el nuevo dataset.

**SVM:** Dicho esto, para el módulo de clasificación “SVM”, hemos tratado de encontrar valores óptimos a los parámetros *SVMType, KernelType* y *degree*. El primer parámetro se encarga de decidir si efectuamos clasificación o regresión. Para esta práctica nuestro objetivo es clasificar las instancias, no efectuar una regresión sobre las mismas, de modo que sólo dos valores de este parámetro han sido tratados. El segundo parámetro mencionado hace referencia al modo de calcular el límite que se encarga de separar espacios para la clasificación de instancias. (Es sabido que el “SVM” tiene parámetros como *Gamma, Un,* etc. Que son aplicados en las fórmulas que guarda el parámetro *KernelType*.). Y el tercer parámetro, es necesario para especificar qué grado de polinomios se tratarán para generar las funciones de separación de espacios que utiliza “SVM”.

El resto de parámetros los hemos dejado con sus valores por defecto, ya que hemos considerado que lo importante es saber qué grado de polinomios tratar para poder crear límites en forma de curva o recta, por ejemplo; qué método de clasificación de entre los dos que puede utilizar “SVM” utilizar; y qué fórmula de generación de límites es la más idónea para esa clasificación.

**RANDOM FOREST:** Respecto al modulo de Random Forest, hemos tratado de optimizar tres de sus parámetros dependiendo del tipo de fichero de entrada que dispongamos. Resulta obvio que si se trata de una gran cantidad de datos, el coste computacional se verá gravemente incrementado. Pero esto solo pasara a la hora de crear el clasificador. Los parámetros estudiados son los siguientes:

-Número de árboles: El algoritmo de Random Forest, consiste en crear varios árboles de decisión, contando con una parte de los datos de entrenamiento para cada árbol. Con este parámetro, estudiaremos cual es la cantidad óptima de arboles respecto a nuestro dataset.

-Nivel de profundidad: Se busca conocer el nivel de profundidad que tendrán los arboles.

-Cantidad de *features*: Cantidad de atributos que tendrán en cuenta cada árbol.

Para la elección de estas características, nos hemos basado en una medida que trae implementada el propio algoritmo, haciendo alusión al dicho, “no necesidad de reinventar la rueda”. Esa medida se denomina *OutOfBagError*.

**BAYES NET:** Con el clasificador BayesNet tuvimos que experimentar con ficheros más simples para comprobar que funcionaba correctamente. Utilizamos los ficheros “hepatitis” y “ScatteringNeutrons”. De esta manera conseguimos comprobar que podíamos seguir con la implementación de otros módulos.

**ONE CLASS:** Primeramente, para hacer las pruebas con el clasificador “OneClassClassifier” reducíamos el número de instancias mediante el módulo “Resmple.jar” de modo que la diferencia entre el número de instancias con valor positiva de clase e instancias de valor negativa de clase fuese la menor posible. Una vez hecho esto, pasábamos el clasificador, que entrenaba únicamente con una clase de ambas.

En las pruebas realizadas sin reducir e igualar los valores de la clase, vimos que fallaba un porcentaje alto de instancias que trataba de clasificar como la clase minoritaria cuando pasábamos el dev. Esto así, cuando aplicamos el “Resample”, comprobamos que los resultados eran bien distintos: Sólo erra en el 0’0345% de los casos.

**NAIVE BAYES:** Vemos que con el “Naive Bayes” conseguimos entrenar un mdelo que devuelve un porcentaje de clasificación correcta del 98’9% sobre el fichero dev.

## 5.-CONCLUSIONES

Se pueden sacar muchas conclusiones de esta práctica. Entre ellas, la más general e importante es la de que hemos trabajado conjuntamente en un trabajo que podría describirse como real. Hemos tenido tres datasets: uno de entrenamiento, otro de evaluación, y finalmente, uno de test, los cuales hemos tenido que usar debidamente (no pasar el test hasta tener los modelos bien entrenados con el train y el dev). Además, los datos estaban ocultos, siendo un claro problema de datos privados o confidenciales, de manera que los nombres de los atributos, así como sus posibles valores, se mostraban de manera que se observaba que habían sido sustituidos por nombres que no eran representativos. Esto nos ha evitado tener la tentación de mirar en exceso los dataset, para saber si los resultados obtenidos tenían sentido o no, y nos ha obligado a tratar de sacar conclusiones de los resultados en base a reglas estadísticas o matemáticas (como pueden ser las probabilidades de estimación de un valor de clase u otro, la frecuencia con que se estimaba una clase u otra, etc.).

Otra de las conclusiones que podemos sacar es que se necesita una idea inicial fija y bien pensada o amueblada para poder trabajar en grupo de manera eficaz y lógica: pensar cómo y cuándo se van a hacer los entregables o los objetivos parciales, pensar cómo podemos unir los distintos módulos generados, cómo representar los resultados, qué parámetros utilizar para no haber incongruencias entre módulos, etc.

Además, nos ha aportado mucho conocimiento el hecho de tener que abordar problemas con distintos clasificadores y sistemas de filtración de instancias, pues al tener que ayudarnos entre todos mutuamente, hemos tenido que preocuparnos de saber y entender las funcionalidades de los trabajos de cada fracción de grupo.

Por último, cabe comentar que los errores y problemas que hemos tenido nos han ayudado para descubrir nuevos método o formas de trabajar con librerías de “weka” (el mencionado problema del “clases not in CLASSPATH!”, por ejemplo).

En definitiva, creo que ha sido una práctica que nos ha gustado, en general, aunque nos dejamos la espinita clavada por no haber conseguido que el SMOTE funcionase como debiese, ni que el SVM lo hiciese para dataset grandes, aún dándole más memoria a la VM de Java.

## 6.- CÓMO INICIAR EL PROGRAMA

URL github: <https://github.com/aorbegozo005/Data_Mining_Practica_5>

Para facilitar usar el programa hemos creado la carpeta “Jarfinalak”. En esta carpeta se encuentran todos los jar necesarios y los scripts sh.

El script está dividido en dos partes:

### Modeloak.sh

Con este script entrenaremos los modelos y se sacarán los mdl con los parámetros optimizados.

#### Parametros:

1-Path fichero train"

2-Path ficherodev

3-(opcional) Posicián de la clase. Si no se especifica se utilizará el último atributo como clase

### Sailkatu.sh

Con este script se utilizarán los modelos creados por el anterior y se harán las predicciones. Por un lado se creara un archivo arff que será la copia del test pero con atributos añadidos de cada clasificador indicando su voto en la instancia.

Después se creará utilizando ese arff un archivo de texto plano indicando la predicción de cada instancia. El método utilizado ha sido el de votación por lo que la clase será el valor más votado entre todos los clasificadores.

#### Parametros:

1-Path fichero test

2-Path fichero donde guardar los resultados intermedios. Un arff con los votos de cada clasificador

2-Path fichero texto plano donde se guardará el resultado final teniendo en cuenta todos los clasificadores

### Completo.sh

Este script simplemente es la suma de los dos anteriores. Ejecuta primero modeloak.sh y luego sailkatu.sh.

Los parámetros son los la suma de los de modeloak.sh y sailkatu.sh con la diferencia de que el la posición de la clase esta al final del todo (Este parámetro es opcional):

1-Path fichero train

2-Path ficherodev

3-Path fichero test

4-Path fichero donde guardar los resultados intermedios.

5-Path fichero texto plano donde se guardará el resultado final teniendo en cuenta todos los clasificadores

6-(opcional) Posición de la clase. Si no se especifica se utilizará el último atributo como clase

#### Completo\_SVM.sh

Los anteriores no usan SVM ya que puede fallar con archivos grandes. En caso de querer usarlo también se debe usar este script. Los parámetros son los mismos que el completo.sh pero todos los parámetros son obligatorios.

## 7.- NOTAS, COMENTARIOS Y ADVERTENCIAS

**SVM:** No entendemos por qué, en el módulo del clasificador SVM salta la excepción de “Java Heap Space” indicando que la MV de Java no tenga la memoria suficiente para hacer frente a la cantidad de datos cargados. Hemos probado hasta dándole 7GB de memoria, pero sigue saltando el error. No obstante, hemos probado a ejecutar el jar con ficheros de menor tamaño como “*diabetes.arff”*, y no hemos tenido ningún problema.

Otro aspecto a tener en cuenta es que tampoco hemos conseguido cargar debidamente el modelo del SVM para pasar el test, pero para remediarlo, hemos decidido que al conseguir el modelo del clasificador también se guarden los valores óptimos de los parámetros en un txt y que a la hora de pasar el test, al script encargado de esta tarea, se le pasen los valores obtenidos y vuelva a entrenar con esos valores para pasa el test. Sabemos que no es una técnica muy ortodoxa, pero hemos comprobado con ficheros como el anteriormente mencionado “*diabetes.arff”* y nos han salido muy buenos resultados, sin problemas.

**SMOTE:** Con el fichero “Smote” también lo hemos intentado todo, pero sigue saliéndonos el error de “Java Heap Space”, y no hemos dado con la tecla para solucionarlo, pero, como ya hemos mencionado con anterioridad, como hemos implementado el módulo “Resample”, creemos que tampoco resulta tan necesario este filtro.