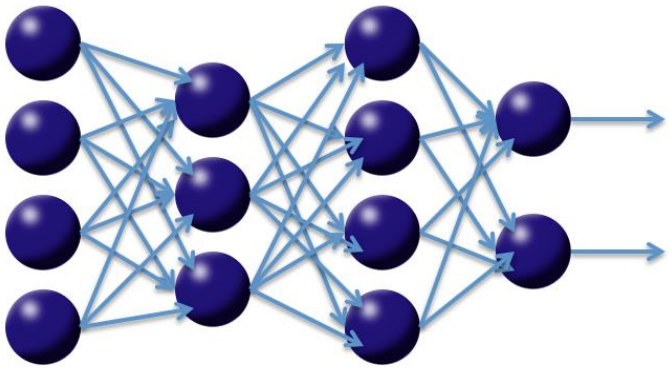
2015

kaggle

otto group

Kaggle Otto Group

MODELOS DE PREDICCIÓN



Carlos Jesús Fernández Basso
cjferba@gmail.com
Ismael González Marín
ismagm@gmail.com

Índice

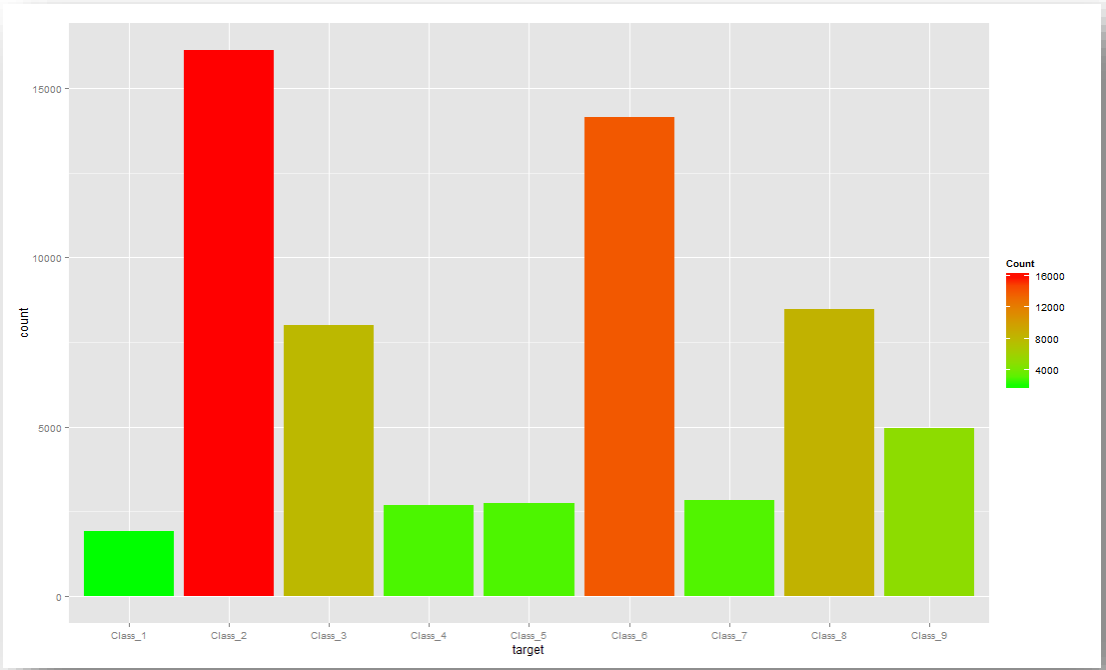
Introducción

Otto Group es una de las mayores compañías de comercio electrónico del mundo que se dedican a vender todo tipo de producto. El problema que plantean es que muchos de estos productos que son idénticos se llegan a clasificar de diferente forma, por lo que necesitan conseguir clusters cuyos elementos sean lo más parecido posible.

El conjunto de datos suministrado está compuesto por más de 200.000 productos, dividido en un conjunto para entrenar los modelos propuestos y otro para realizar las predicciones. A continuación veremos los análisis de los dos conjuntos de datos.

Train

El conjunto de entrenamiento lo componen 61878 productos y cada uno de ellos tiene 95 características. Una de ellas es la clase a la que pertenece el producto de las nueve posibles. A continuación se muestra el histograma de la distribución de clases en el conjunto train y cómo podemos ver, las clases están desbalanceadas.



Test

El conjunto de entrenamiento lo componen 144368 productos, cada uno de ellos con 94 características. Como podemos ver el conjunto de predicción es prácticamente el doble que el conjunto de entrenamiento.

Características

En este apartado vamos a ver las 94 características de forma resumida. Primeramente empezaremos viendo que la primera característica es el *id* de los productos. Éste no posee información relevante para la predicción de los elementos, por lo que eliminaremos esta característica para nuestros análisis.

En el resto nos encontramos que son valores enteros, con un rango de valores que depende de la característica. Aquí podemos ver algunas de ellas y el número de posibles valores que toman.

Característica 2: 42 valores

Característica 3: 37 valores

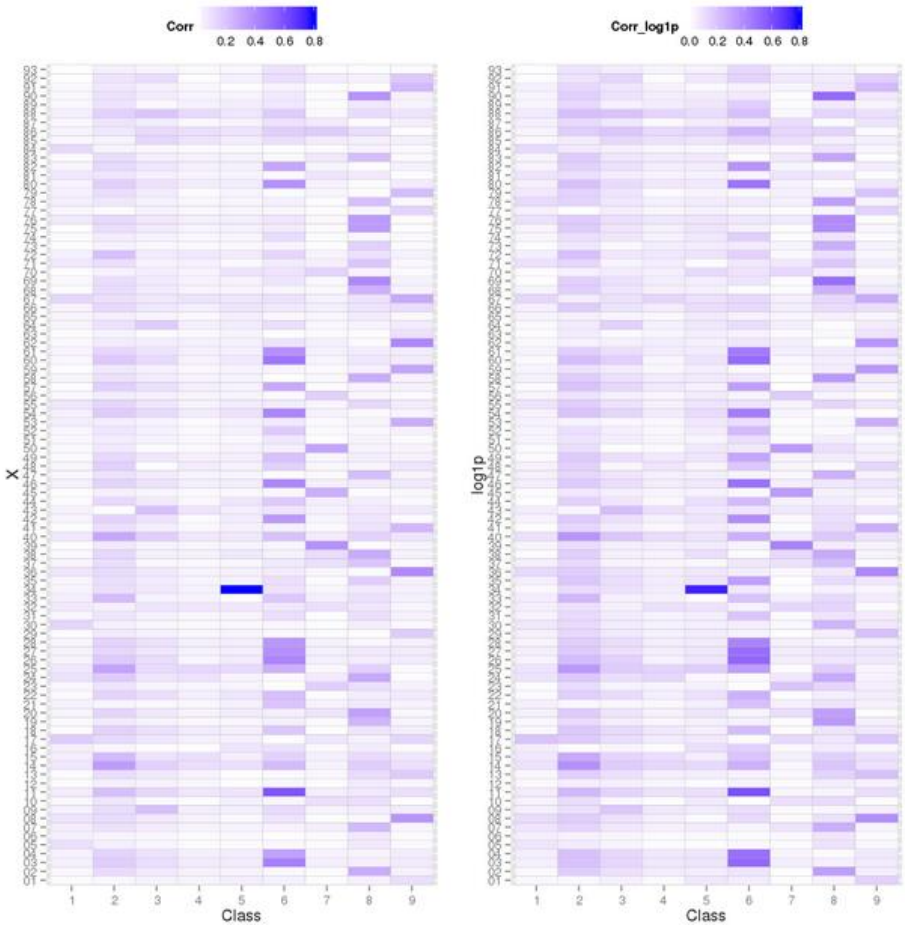
Característica 4: 48 valores

Característica 5: 59 valores

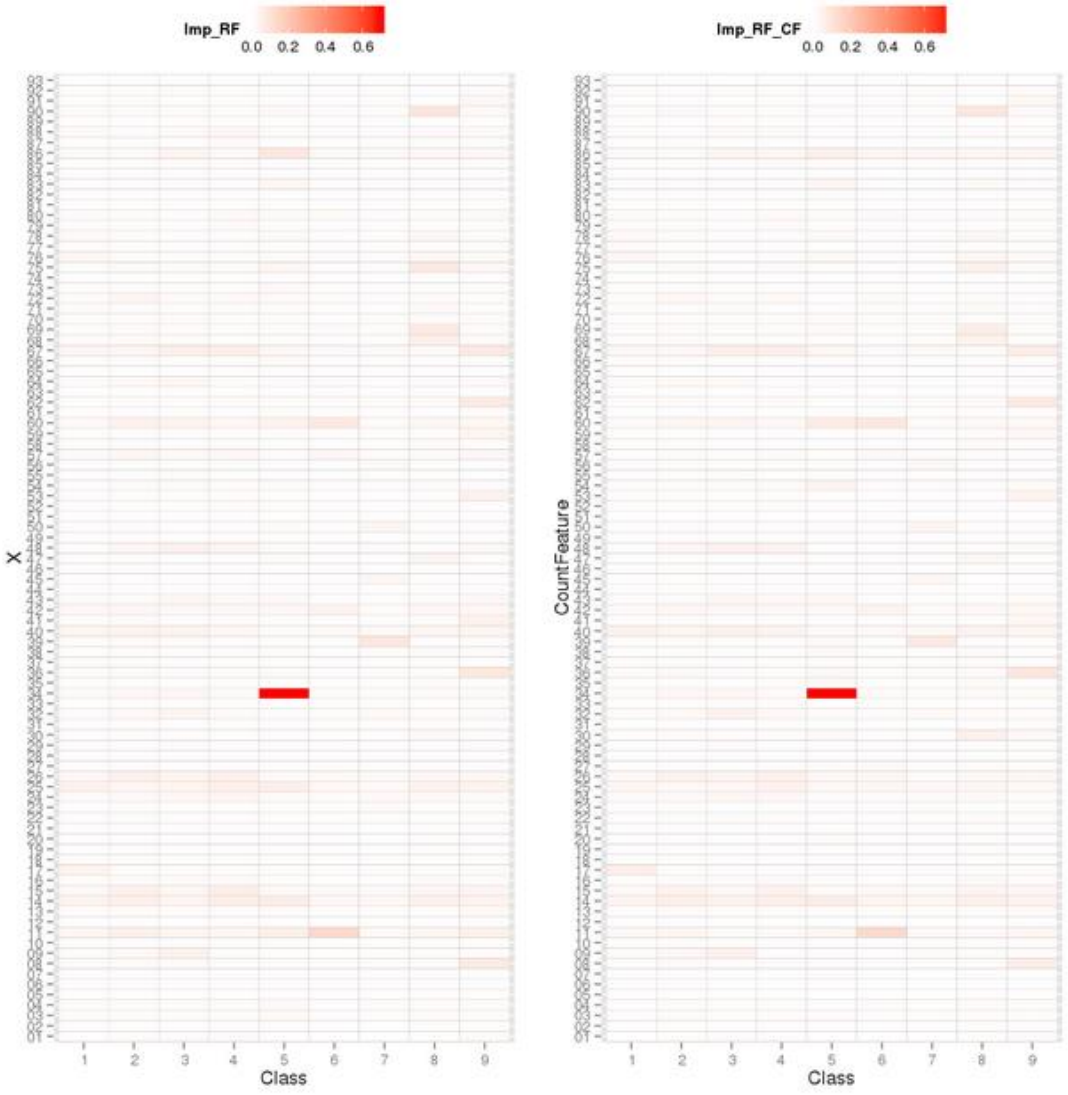
Característica 6: 15 valores

Característica 7: 9 valores

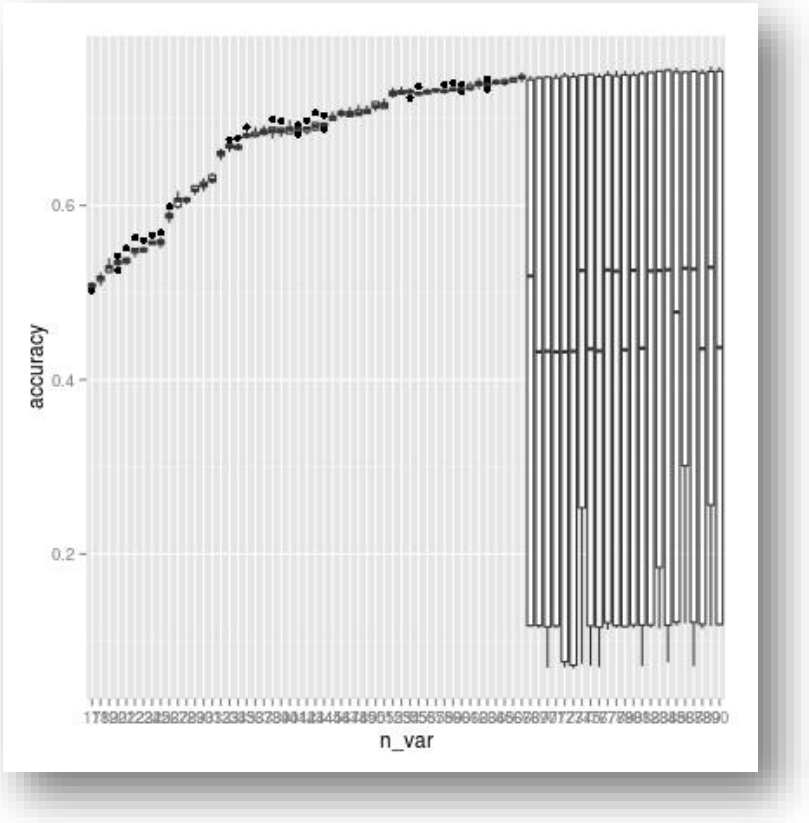
Vamos a ver ahora la correlación e importancia de las características en las diferentes clases, análisis que creímos importante viendo que 93 características nos parecían muchas para realizar la clasificación.



En esta gráfica, la importancia está generada mediante el algoritmo Random Forest.



Aunque estas gráficas nos muestran que algunas variables no tienen relevancia, las probamos todas mediante recurssive feature elimination con Random Forest como clasificador base y usando validación cruzada. En la siguiente gráfica podemos ver cómo el error era mayor a partir de añadir 65 variables. Por ello, guardamos este resultado como posible selección de características.



Método de evaluación

Para la evaluación de los resultados se utilizó la siguiente métrica que proponía la propia competición en Kaggle [6], usando logaritmo neperiano en vez de en base diez.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij}),$$

Donde N es el número de elementos, y_{ij} es 1 cuando la clase predicha coincida con la real y cero en caso contrario, y p_{ij} es la probabilidad predicha por nuestro método de clasificación para cada clase.

Antes de realizar la métrica los datos se trasforman de esta forma para que el error esté acotado inferiormente a -34.53 cuando la probabilidad es igual a cero.

$$max(min(p, 1-10^{-15}), 10^{-15})$$

Preprocesamiento

Trasformación

Para algunas técnicas, como las redes neuronales, debemos trasformar los datos para que se distribuyan en un rango de valores pequeño, por lo que hemos propuesto y probado diferentes técnicas.

Scale:

En este caso realizamos una normalización z-score. Es una normalización muy conocida en estadística y para una primera aproximación era la mejor opción.

$$z = \frac{x - \mu}{\sigma}$$

Log:

La siguiente trasformación fue el logaritmo, en este caso. Como el logaritmo de o no está definido, lo que se realizó exactamente fue $\log(x + 1)$.

Logit:

Funciona muy bien en modelos de redes neuronales, por lo que nos pareció buena opción. Ésta se define mediante la siguiente ecuación (para calcular el logaritmo de cero hacemos como hemos dicho en la trasformación anterior).

$$\operatorname{logit}(p) = \log\left(\frac{p}{1-p}\right) = \log(p) - \log(1-p).$$

Tf-idf

En este caso probamos esta trasformación [2] creyendo que podría darnos un resultado diferente a las anteriores:

$$\operatorname{tfidf}(t, d, D) = \operatorname{tf}(t, d) \times \operatorname{idf}(t, D)$$

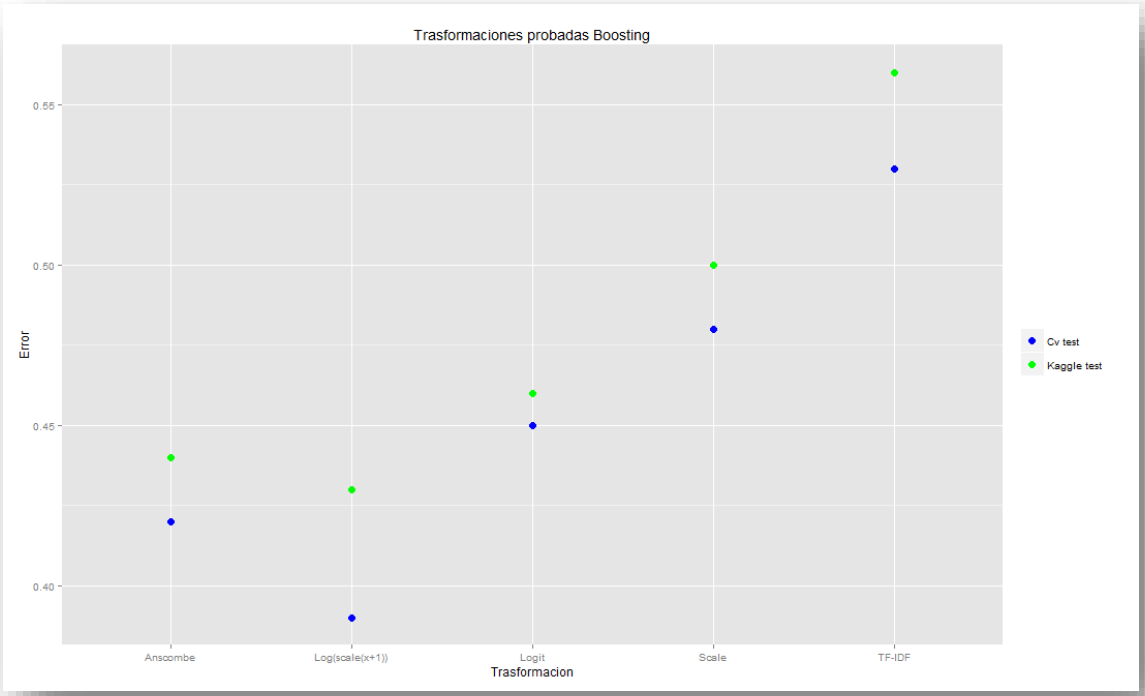
Anscombe

Al ver que la distribución de los datos seguía una distribución de Poisson (lo comentó la gente en el grupo de kaggle), probamos con esta trasformación [3].

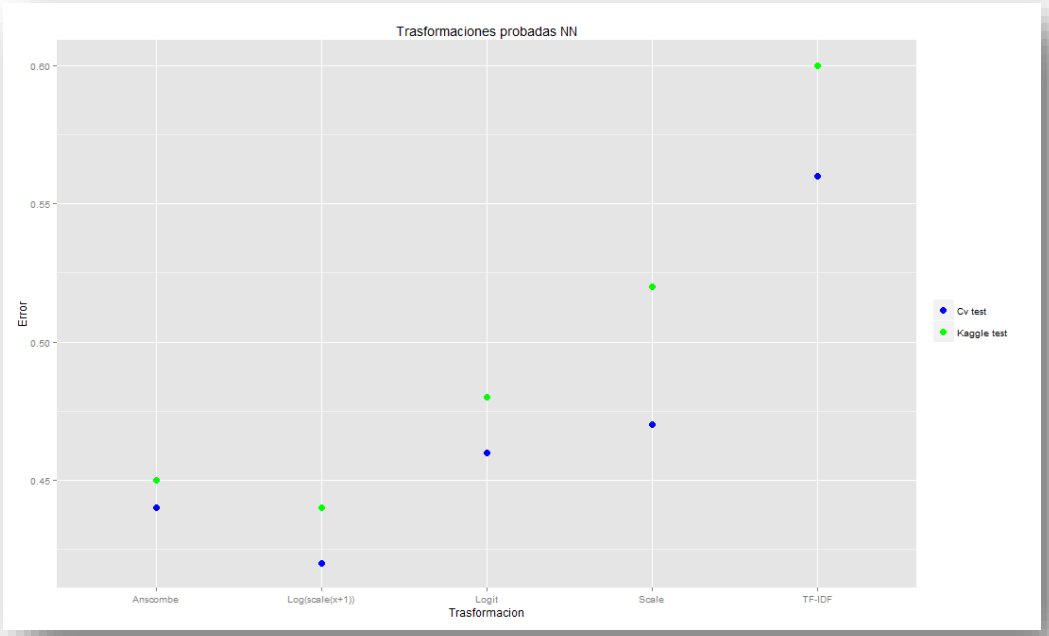
$$x \mapsto 2\sqrt{x + \tfrac{3}{8}}$$

Tuvimos diferentes resultados al probar cada una de las trasformaciones, pero las que mejor resultado obtuvieron fueron las siguientes:

- Utilizando boosting



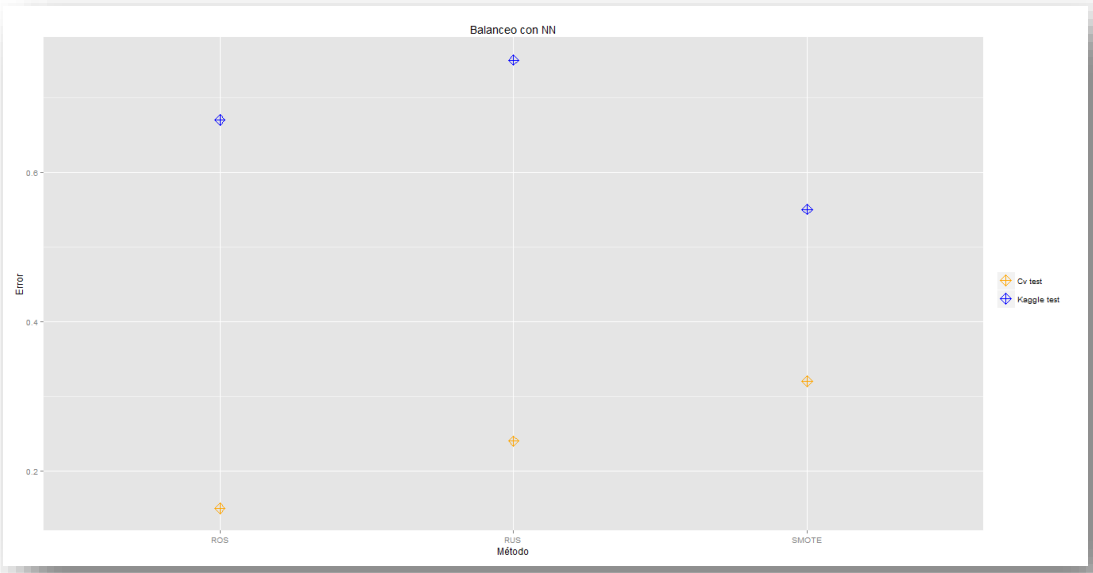
- Utilizando Redes Neuronales:



Métodos de Balanceo

Por último, en nuestro pre-procesamiento, vamos a también intentar tratar el desbalanceo. Para ello se crea un algoritmo SMOTE [4] para multiclase que más adelante se explicará en profundidad.

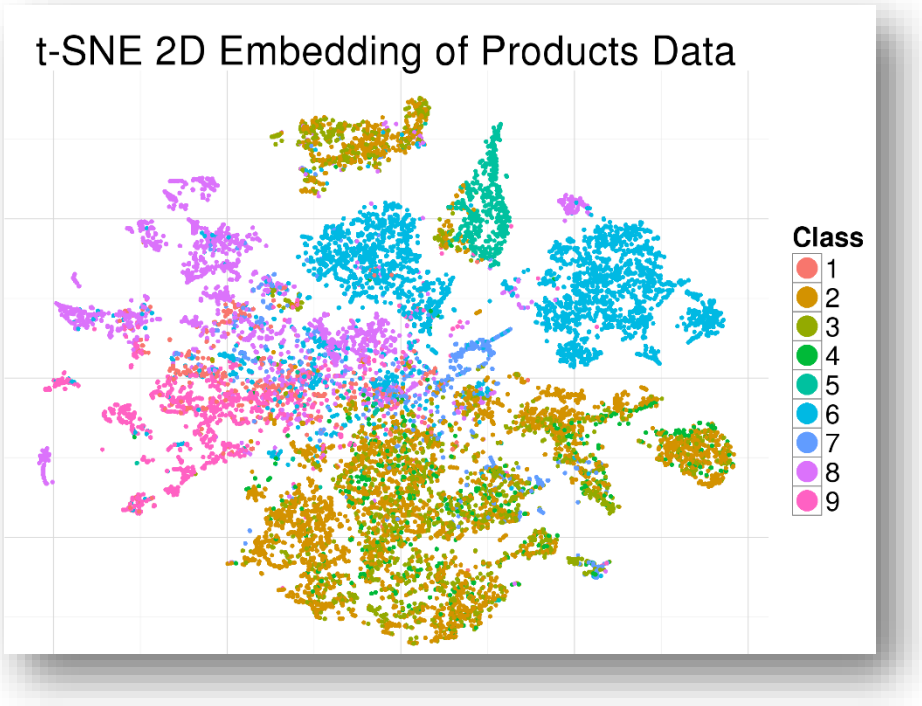
Además de este método también utilizamos métodos más sencillos como random oversampling y random undersampling [5].



Como hemos visto en la gráfica, el balanceo nos produce mucho sobre-aprendizaje, y con métodos como XGBOOSTING el sobre-aprendizaje es mayor. Por lo tanto, el balanceo quedaba descartado.

Reducción de características

También se ha realizado un análisis de componentes principales mediante *PCA* y utilizando *T-sne*. Estos dos métodos nos permitieron ver cómo están distribuidos los datos en una proyección bidimensional y el solapamiento que hay entre las clases. Como podemos apreciar en la siguiente gráfica, y luego veremos en las predicciones, las clases están muy solapadas.



Además gracias al gráfico http://data-projector-visualizations.sourceforge.net/otto/ que realizo Foxtrot, podemos ver cómo están distribuidas las clases en 3 dimensiones. Y como ocurre en el anterior podemos ver cómo la clase 6 (azul) está muy poco solapada. Más adelante veremos que en las predicciones esta clase sale con valores de probabilidad muy altos y tiene en el CV un valor muy alto de acierto.

Algoritmos de clasificación

Para la clasificación de este problema, se probaron varios algoritmos con el propósito de encontrar las mejores configuraciones y pre-procesamiento para cada uno y así poder realizar luego un ensamble con todos ellos con el fin de mejorar el resultado individual que obtenían por separado.

Redes Neuronales

Como en kaggle no se puede usar software propietario realizamos las NN en R y Python.

Redes neuronales en R

En este lenguaje hay varias opciones para implementar redes neuronales. Veremos a continuación los que se han probado y los resultados obtenidos.

NNET

Para R existen varios paquetes que realizan redes neuronales. El primero que usamos fue nnet [7], pero es bastante limitado, por lo que usamos solamente una red neuronal de una capa (Perceptrón). Esta red nos dio un resultado de 0.9, muy lejos de lo que obteníamos inicialmente con Random Forest en CV con balanceo, unos 0.68.

RSNNS

Este es el segundo paquete de R [8] que utilizamos y a diferencia de NNET trae varias redes más, como redes de Jordan, Elman, perceptrón multicapa…

Para nuestro propósito se probó a utilizar la red de Jordan y un Perceptrón multicapa. Cada uno de estos dos experimentos nos dio resultados dispares pero uno de ellos sí proporcionaba mejora sobre el Random Forest que hemos citado anteriormente.

El primer experimento fue con la red de Jordan pero el resultado fue desastroso, puesto que el error generado fue de 1.4, el peor que se había obtenido hasta ese momento.

El segundo fue con el Perceptrón multicapa, con la configuración siguiente:

- Iteraciones máximas:250
- Tamaño:28
- Parámetro de aprendizaje:0.1
- Función de aprendizaje: Rprop

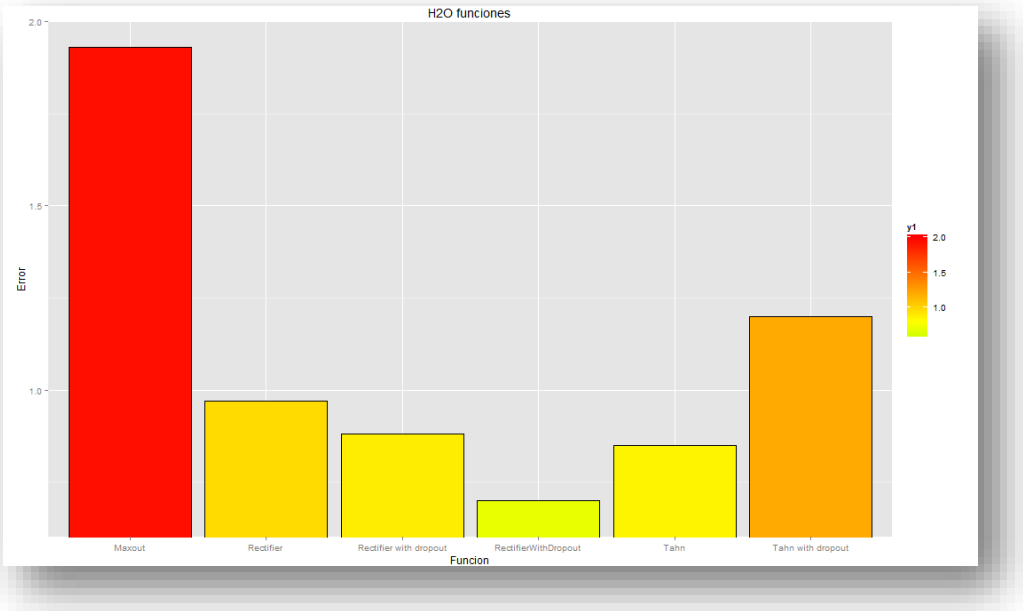
Esto, junto con la trasformación de $scale(\log(x+1)+1)$ dio como resultado 0.62, mejorando al Random Forest.

H2O

Este paquete contiene cantidad de algoritmos de aprendizaje automático. Entre ellos se encuentra uno de redes neuronales denominado DeepLearning.

Deeplearning de H2O [9] se basa en una red neural artificial multicapa feed-forward que se entrena con descenso de gradiente estocástico utilizando back-propagation. La red puede contener un gran número de capas ocultas que consta de neuronas con funciones TANH, rectificadoras y activación MAXOUT. Las características avanzadas tales como la tasa de adaptación de aprendizaje, capacitación impulso, la deserción escolar, L1 o L2 regularización, puntos de control y búsqueda de rejilla permiten una alta precisión predictiva. Cada nodo de cómputo entrena una copia de los parámetros del modelo globales en sus datos locales con multi-threading (asíncrona), y contribuye periódicamente al modelo global a través del modelo de promedio en toda la red. De esta forma el algoritmo es más eficiente y nos permite utilizar el máximo rendimiento de nuestro sistema.

Además este algoritmo nos permite definir varias configuraciones, para poder probar diferentes combinaciones de parámetros. Así se probaron diferentes funciones para las capas como: RectifierWithDropout, Tahn, TahnWithDropout, Rectifier, RectifierWithDropout, Maxout.



Como se puede apreciar en la gráfica la función que mejor resultado nos dio fue RectifierWithDropout. A partir de éste comentaremos el resto de parámetros elegidos.

- La red tienen 3 capas con las siguientes neuronas:

- $hidden\_dropout\_ratio = c(0.5,0.5,0.5)$ *
- $input\_dropout\_ratio = 0.05$

- $epochs = 120$
- $l1 = 1e-5$*
- $l2 = 1e-5$*
- $rho = 0.99$*
- $epsilon = 1e-8$*
- Ejemplos de entrenamiento por iteración = 2400

* *Estos parámetros son escogidos de los ejemplos de la web https://leanpub.com/deeplearning/read*

Los datos utilizados en este algoritmo están trasformados con Anscombe, tal y como comentamos en la sección anterior.

Con todo esto el mejor resultado de esta red neuronal fue **0.453,** el cual nos da una gran mejora frente a Random Forest.

Además de esto, realizamos otra prueba con selección de características y con balanceo, la cual nos mejoró nuestro resultado a **0.4437.**

Redes neuronales en Python

Viendo que una de las librerías más utilizadas en machine learning es Scikit-learn, de Python, decidimos probar y aprender su funcionamiento, ya que tiene algoritmo clásicos como RF, boosting… aparte de varios módulos de redes neuronales. Nosotros utilizamos dos de ellos: LASAGNE y THEANO.

Estas redes se denominan Redes neuronales convolucionales, con una gran aceptación en el tratamiento de imágenes. Este tipo de red tiene varios tipos de neuronales, pero nosotros utilizaremos las capas densas (Dense Layer) y las capas de Dropout (Dropout Layer).

Después de comentar esto, para los experimentos se utilizaron diferentes configuraciones. Dos de ellos nos dieron resultados relevantes y fueron escogidos como resultados para realizar un ensamble.

1. La primera red que nos dio un resultado de **0.461** tenía la siguiente estructura.
    a. 3 capas ocultas con 520, 460 y 160 neuronas.
    b. Coeficiente de aprendizaje es 0.001.
    c. Actualización de las neuronas mediante adagrad.
    d. Trasformación de los datos con Anscombe.

2. La segunda red tiene las siguientes características y dio como resultado **0.445**:
    a. Dos capas densas y entre ellas capas de Dropout, con 1000 neuronas la primera y 500 la segunda.
    b. Los coeficientes de Dropout de las capas son 0.25 y 0.18
    c. El resto de parámetros son como en la primera.

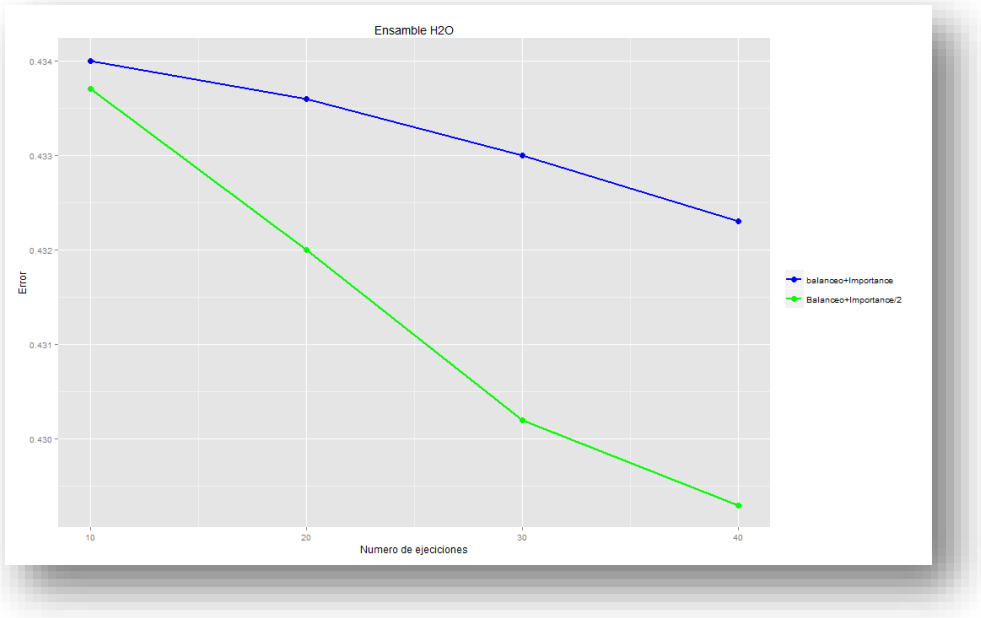
Ensamble de Redes

Después de ver los resultados de estas redes, y al ser la salida una matriz de 140000 filas (elementos del test) y 10 columnas (id, clase1, clase2...clase9) con una probabilidad para cada clase, probamos a realizar en cada red varias ejecuciones y usar la media geométrica y la aritmética de los resultados que se obtenían. A continuación veremos las mejoras sobre las dos mejores propuestas (H2O y Python).

H2O

Primero probamos 10 ejecuciones de la red normal (sin balanceo ni selección de características) dándonos un resultado de **0.44**. Al ver esta mejora, nuestra segunda prueba se basó en ese mismo método pero con balanceo e importancia de variables, el cual mejoró a **0.434**.

Viendo esto se realizaron diferentes pruebas con este segundo mejor resultado, con el fin de intentar mejorarlo al máximo.

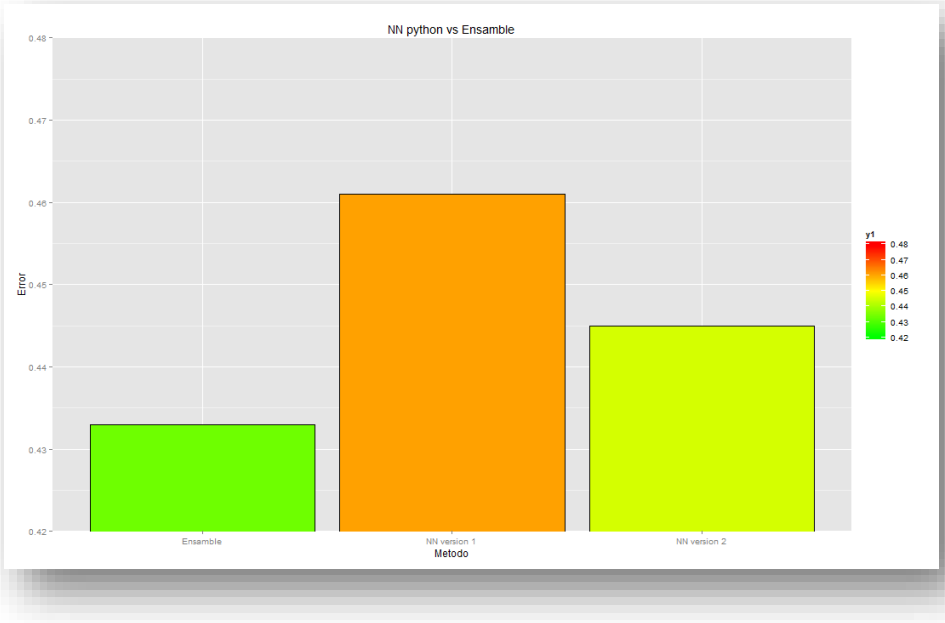


Como se ve en el gráfico, funcionaba mejor el ensemble si de numero de iteraciones solo realizabamos la selecion de caracteristicas en la mitad. Tambien comentar que a partir de 40 ejecuciones no se ganaba nada e incluso empezaba a empeorar, y como era muy costoso en tiempo preferimos quedarnos en esa cantidad.

Python

Viendo la mejora que se produjo en el algoritmo anterior, realizamos la misma técnica de ensamble con Python. En este caso, al ver que métodos un poco diferentes daban diversidad a la solución y mejoraban el resultado, optamos por utilizar la mitad de resultados con una red de Python y la otra mitad con la otra.

Se realizó en ensamble con 20 ejecuciones (10 de cada red), obteniendo la siguiente mejora:



Por ultimo comentar que la realización de los dos ensambles que se muestran es mediante la media aritmética. Como después de experimentar con la media geométrica y aritmética pudimos ver que no había diferencias significativas (mediante Test de student), decidimos escoger la aritmética.

XGBOOSTING, Random Forest y SVM

XGBOOSTING ha sido nuestro clasificador de referencia porque es muy robusto frente a conjuntos desbalanceados. El gran inconveniente que presenta es que utiliza un número muy elevado de parámetros y algunos de ellos creemos que son sensibles al número de instancias presentes en el conjunto de entrenamiento.

Este método en solitario es el que nos da mejores resultados, llegando a **0.431** (igualando a los ensambles de NN). Estudiando sus resultados comprobamos que las probabilidades en las clases poco balanceados son muy bajas (tiende a dividir la probabilidad entre los 3-4 más probables) lo que produce mucho error.

Random Forest también se usó como algoritmo de referencia, sobre todo en la segunda mitad de competición ya que, en nuestro caso, era mucho más rápido que SVM para el OVO y en solitario conseguimos que funcionara mejor que SVM, siendo este uno de los algoritmos con el que comenzamos por ser uno de los denominados “estrella”.

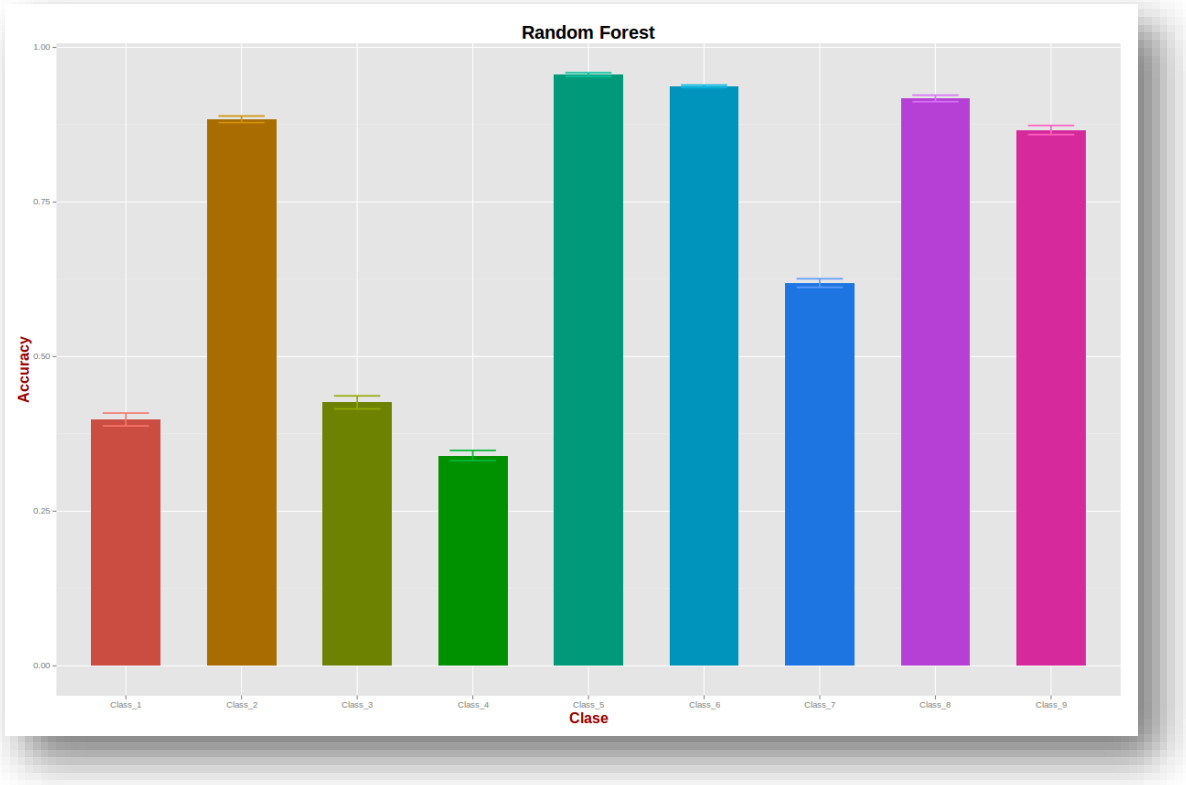
A continuación vamos a verlo todo en conjunto

Con Random Forest teníamos como punto de partida el benchmark propuesto por la propia organización y cuyo código se puede encontrar en GitHub. Aunque está escrito en Python, se puede ver que usa solamente diez árboles y que el número de variables candidatas a ser seleccionadas de forma aleatoria en cada división es el que trae por defecto. El resultado que se alcanza de esta forma es de **1.5387** según la propia validación de Kaggle.

Haciendo primeramente la transformación $y = \log(x + 1)$, ampliando el número de árboles a 1000 y usando 3 variables en cada división, nuestro error disminuye hasta $\mathbf{0.6880 \pm 0.0023}$ en una validación cruzada 5-fold (que será nuestra forma de validar internamente nuestros resultados, destinando el ochenta por ciento de los datos a entrenamiento). La precisión que se obtiene es un $\mathbf{0.7922 \pm 0.0014}$, con una distribución por clases de la siguiente forma:

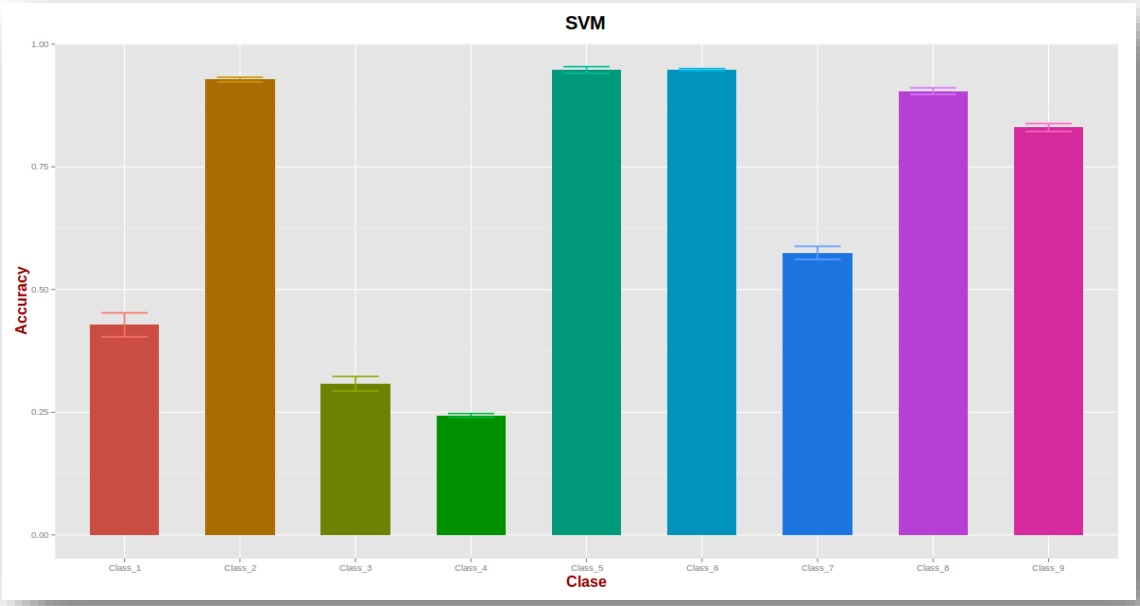
	Mean.Accuracy	Sd
Class_1	0.3983019	0.010407332
Class_2	0.8836811	0.005305709
Class_3	0.4259428	0.010478153
Class_4	0.3400000	0.008294590
Class_5	0.9559097	0.002991734
Class_6	0.9369179	0.001994033
Class_7	0.6189622	0.006890707
Class_8	0.9174651	0.005205630

Gráficamente podemos ver esa distribución de clases como sigue:



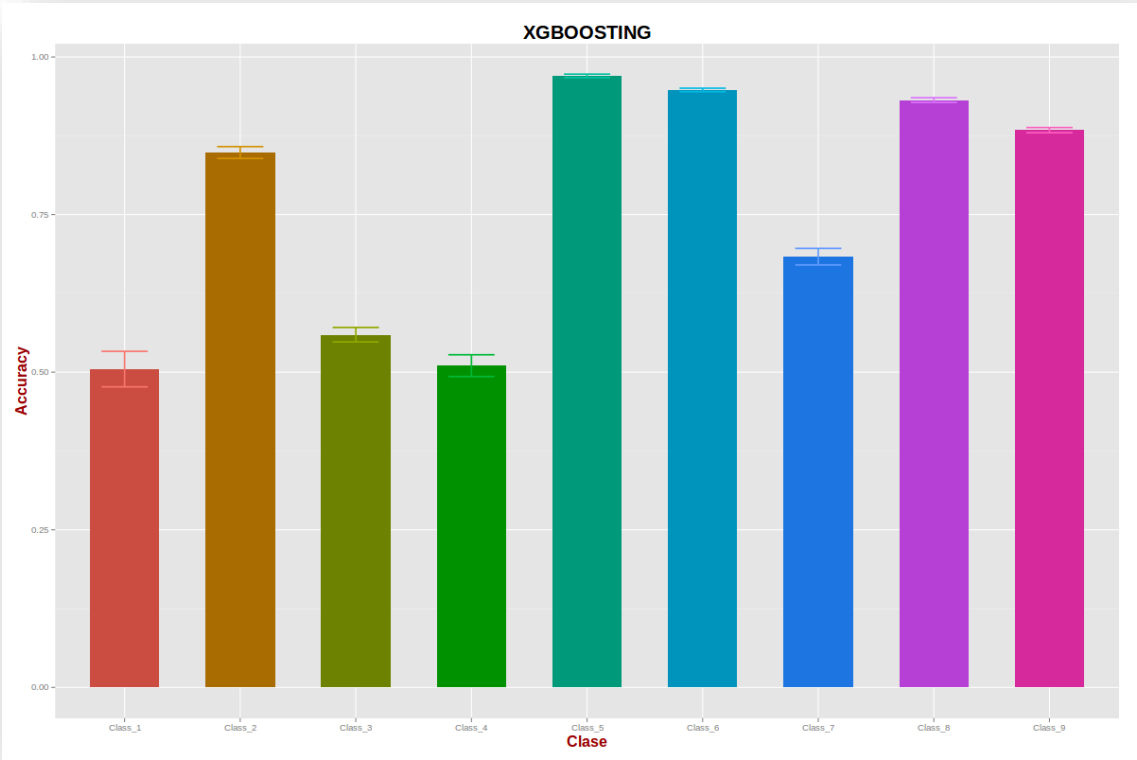
El SVM que se usó era el del paquete e1071, con kernel polinomial e hiperparámetros determinados automáticamente por el propio algoritmo. El error que se obtenía era de 0.591 ± 0.009 mientras que la precisión fue de 0.7807 ± 0.0020.

Su correspondiente distribución por clases es así:



Perdíamos precisión de forma apreciable en la clase 4, 3 y 7 aunque ganábamos en la 1 y en la 2, la mayoritaria.

Si lo comparamos con nuestro mejor clasificador en solitario, el XGBOOSTING, con un error de 0.476 ± 0.009:



Es decir, que nuestro mejor clasificador no predecía la mayoritaria tan bien como Random Forest o SVM pero presentaba un mejor comportamiento en el resto, sobre todo en la 1 y la 4, que eran las que estaban más desbalanceadas.

A partir de aquí planteamos un par de estrategias de preprocesamiento partiendo de la misma transformación.

Algoritmos de preprocesamiento

Iterative-Partitioning Filter

La primera sería el filtrado del ruido en el conjunto de entrenamiento, mediante el algoritmo Iterative-Partitioning Filter. En vez de usar un único clasificador utilizamos tres: LMT (Logistic Model Trees), J48 y Bagging. Para los dos primeros se utilizó el paquete RWeka y para el último adabag. Generando once clasificadores y usando mayoría de 6 sólo conseguíamos eliminar 4 instancias, mientras que para cinco clasificadores y mayoría de 3 no se eliminaba ninguna. Con la transformación $scale(\log(x+1)+1)$ se llegaban a eliminar alrededor de unas 600 instancias, pero finalmente no se adoptó esta estrategia y se utilizó el conjunto sin filtrar durante el resto de la competición.

SMOTE y Tomek Link

La otra estrategia era limpiar un poco las clases con Tomek Link y realizar balanceo con SMOTE. Los paquetes de los que dispone R no permiten realizar este tipo de preprocesamiento para multiclase, así que tuvimos que realizar nuestra propia implementación. En el primer caso nos apoyamos en el paquete *unBalanced* y en su función *ubTomek*. Ésta permite hacer un Tomek Link siempre y cuando la clase mayoritaria esté etiquetada como 0 y la minoritaria como 1. Así que planteamos una estrategia One vs. One en la que por cada pareja de clases identificábamos a la mayoritaria, la reetiquetábamos y luego aplicábamos la función como si se tratara de un problema binario.

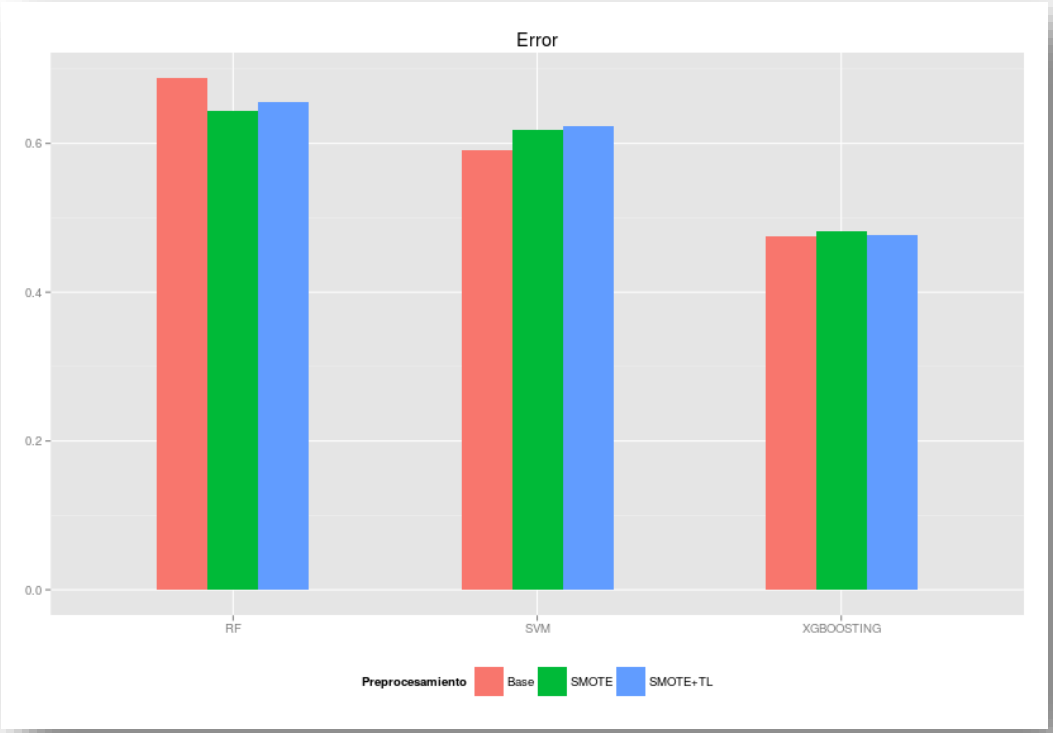
Con SMOTE optamos por una estrategia sencilla consistente en obtener el número de instancias de la mayoritaria y generar el suficiente número de sintéticos que permita igualar en número de instancias el resto de las clases. *unBalanced* permite hacer un SMOTE con la función *ubSMOTE*, pero su implementación no permite fijar de antemano la proporción de minoritarios y mayoritarios que se crean, sino que funciona en base a un sistema de ratios entre creación de instancias minoritarias y selección de mayoritarias según el desbalanceo entre uno y otro, lo que impide que se pueda saber de antemano la cantidad de uno y de otros que se van a generar (incluso probando valores de los ratios llega a ser complicado adivinar si crearán más o menos de una clase o de otra).

Por tanto, decidimos utilizar parte de una implementación facilitada por Chris Cornelis en su asignatura de Minería Avanzada que, aunque es menos eficiente en comparación con la del paquete ubSMOTE, nos permite mayor control y flexibilidad, pudiendo disponer de multicore para acelerar la generación de sintéticos.

Un detalle a tener en cuenta es que para si balanceamos y queremos hacer la validación cruzada hay que tener un poco de cuidado, puesto que primero hay que

particionar los datos en entrenamiento y test y luego aplicar el preprocesamiento solamente a la de entrenamiento. Esto implica que hay que realizar esta operación cinco veces, lo que conlleva un tiempo considerable.

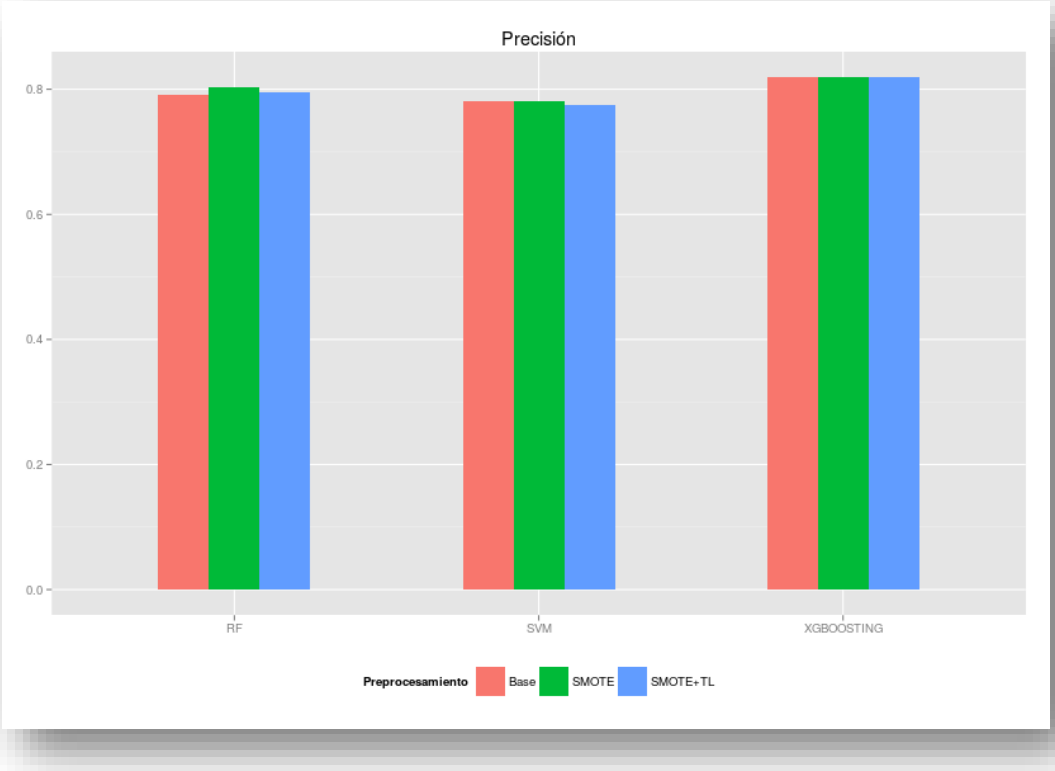
Los resultados de los tres algoritmos anteriores tanto con el clasificador base como con SMOTE y SMOTE+TL se pueden comparar en la siguiente gráfica:



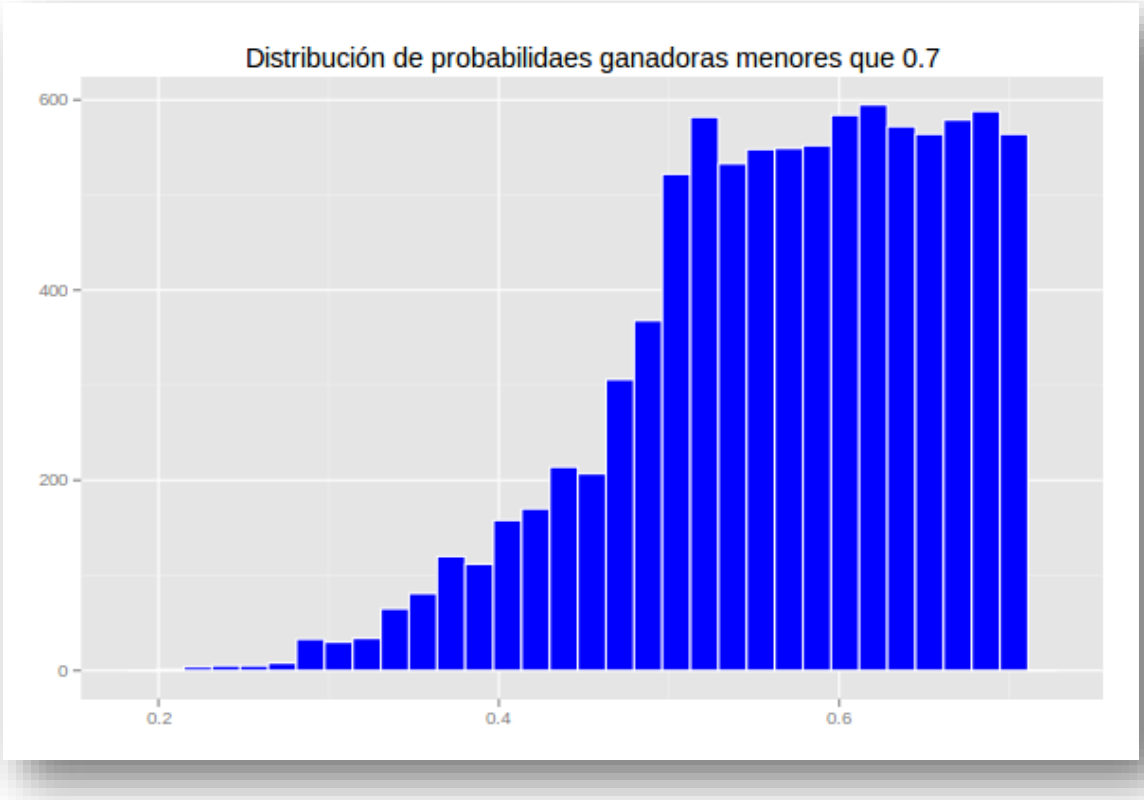
Se puede apreciar que usando Tomek Link después del balanceo el resultado es peor en general tanto en precisión como en error respecto a SMOTE. También se ve que SVM funciona mejor sin este preprocesamiento, que XGBOOSTING es bastante robusto ante él (aunque funciona mejor sin balancear) y que el único que se beneficia es Random Forest.

En definitiva, en las primeras semanas de competición una de nuestras apuestas fue por SVM, llegando a conseguir como mejor resultado **0.62761** en Kaggle con la transformación planteada al principio, mientras que Random Forest ni se llegó a probar por la mala cv que obteníamos. Conformé avanzó la competición y descubrimos que la transformación $scale(\log(x+1)+1)$ estaba dando buenos resultados con XGBOOSTING, probamos de nuevo con SVM y el mismo kernel polinomial. El error fue de 0.56428 sin balanceo. Posteriormente, cuando intentamos la modificación de las probabilidades a mano por primera vez como forma de postprocesamiento, hicimos una única subida usando Random Forest con SMOTE y usando 1000 árboles para tener una referencia (no se modificaron las probabilidades), obteniendo un **0.549699**.

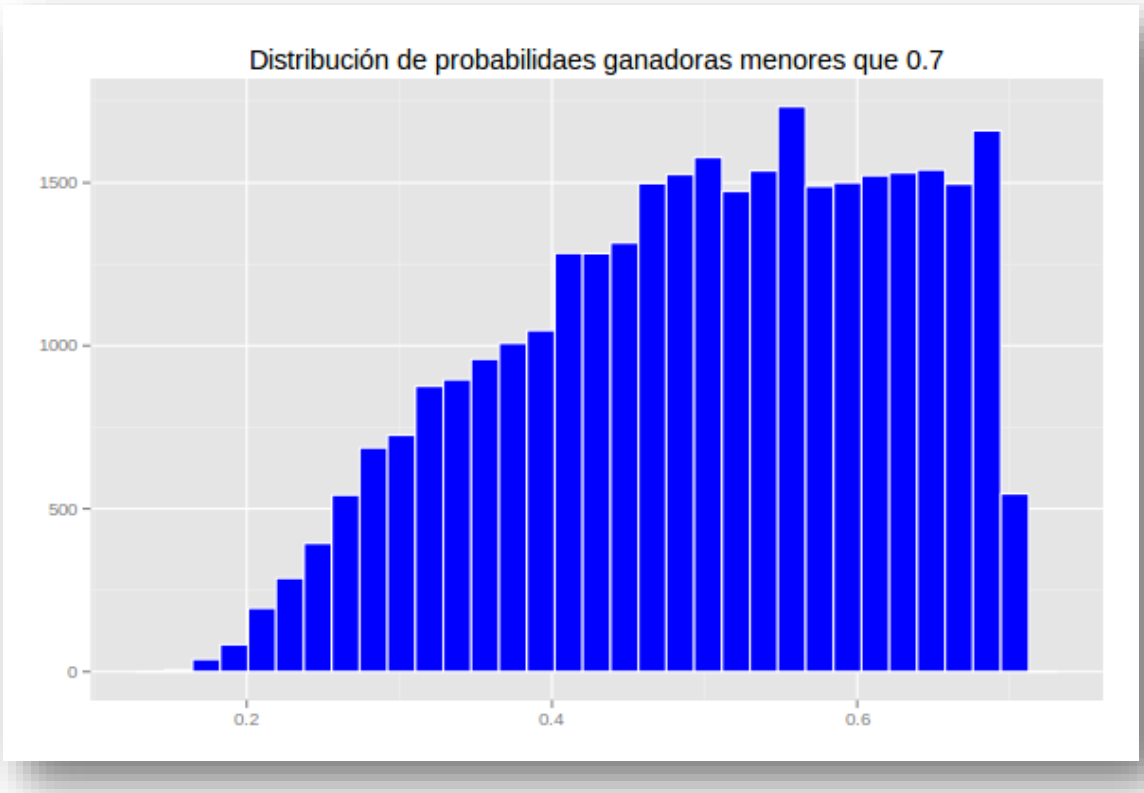
Vistos los pobres resultados, decidimos centrarnos en XGBOOSTING hasta que



planteamos usar One vs. One con Random Forest como algoritmo de experimentación. Finalmente no se adoptó este enfoque con el método que más adelante explicaremos porque al ser incapaces de obtener probabilidades perdíamos toda la potencia del XGBOOSTING, que se puede entender fácilmente con la siguiente gráfica:



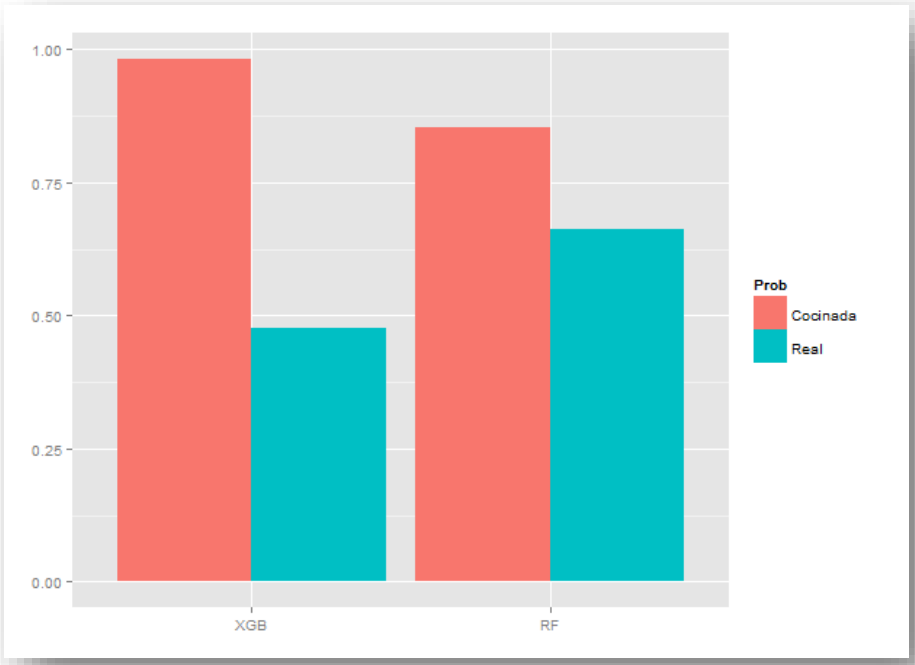
Mientras que la siguiente es para el caso de Random Forest:



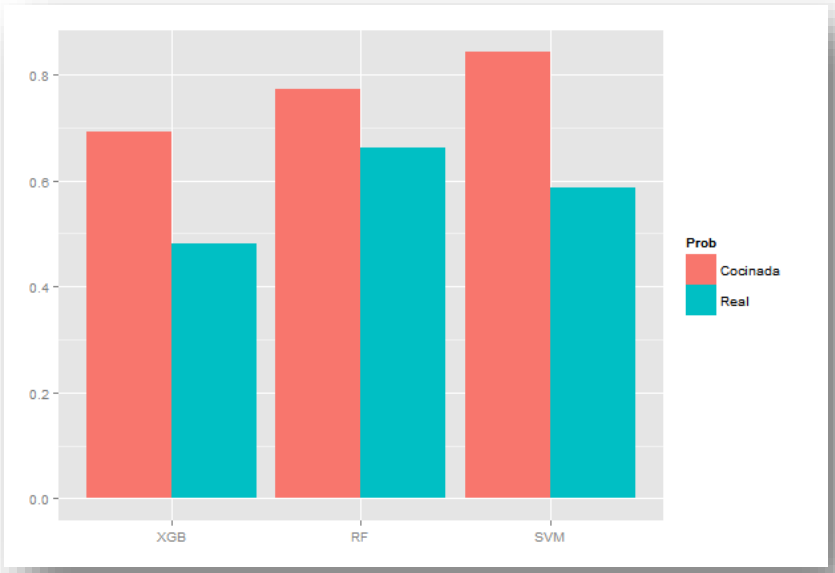
Se puede observar claramente que Random Forest es más indeciso o conservador que XGBOOSTIN a la hora de asignar la probabilidad de la clase ganadora, por lo que al final penaliza en el cálculo del error.

"Probabilidades cocinadas"

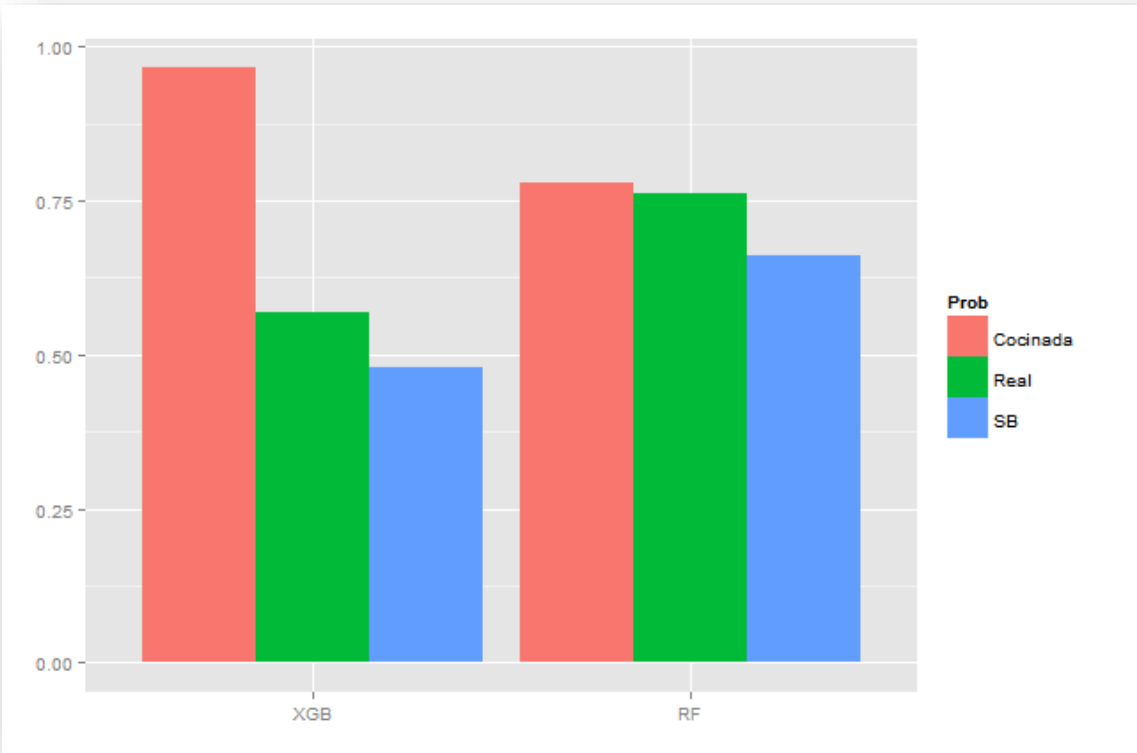
Tras la clase en que se plantea intentar modificar las probabilidades que da la predicción para intentar reducir el error, probamos a hacer una subida a Kaggle cambiando la probabilidad de la clase ganadora directamente a uno, usando como clasificador Random Forest y obteniendo como resultado **1.09921**. A continuación realizamos una serie de pruebas con nuestro conjunto completo de entrenamiento, utilizando una única partición como entrenamiento, dejando el veinte por ciento de los datos como test y usando siempre la misma a lo largo de una serie de pruebas. Las primeras fueron con XGBOOSTING y Random Forest cambiando la probabilidad de la ganadora a 0.9 y de la segunda clase a 0.1.



Como XGBOOSTING se muestra un poco indeciso con algunas instancias (hemos visto que hay un número considerable de ellas en que la clase ganadora tiene una probabilidad inferior a 0.7) decidimos utilizar cambiar únicamente aquellas instancias cuya probabilidad de la clase mayoritaria fuera superior a 0.7, modificando las probabilidades a 0.8 para la primera clase y 0.1 para la segunda y tercera:



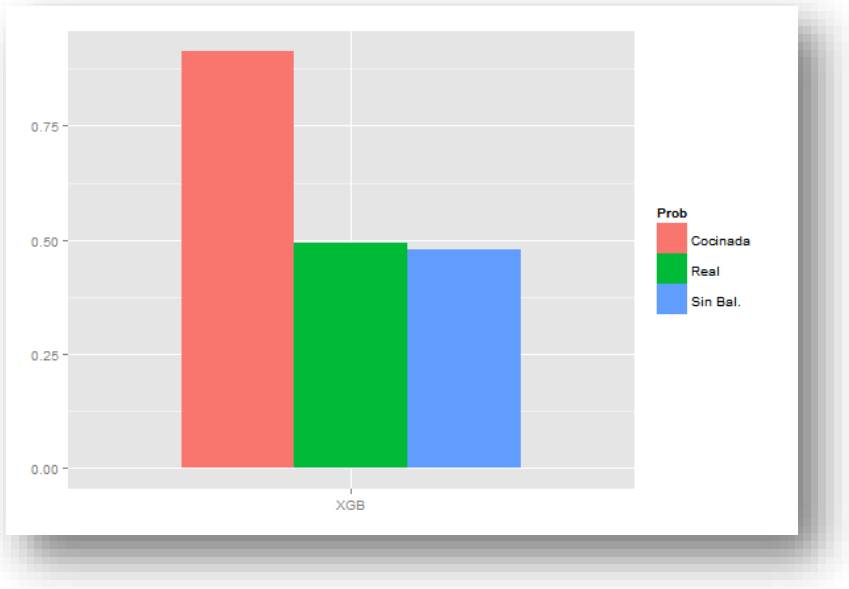
El resultado mejora pero dista mucho del que ofrece el propio algoritmo. Aun así decidimos probar con algunos métodos de balanceo que no habíamos tenido en cuenta previamente, para ver si mejoraba respecto al SMOTE que ya habíamos intentado. Por ejemplo, en la siguiente figura mostramos los resultados de usar Random Undersampling. En azul se representa el error que obteníamos con ese algoritmo sin balanceo, en verde con Random Undersampling y en rojo las probabilidades modificadas tras el propio



Random Undersampling:

Usamos también SVM, pero el resultado fue tan desastroso que no lo pintamos porque nos fastidia bastante la escala. En ese caso para la real obtuvimos un 4.122258 y para la modificada 13.67191.

También probamos Random Oversampling, aunque tras probarlo con XGBOOSTING y ver que no mejoraba ni siquiera al SMOTE no perdimos el tiempo con otros algoritmos:



En definitiva, no era posible modificar las probabilidades a mano sin más. Pudimos comprobar que existía un número considerable de veces en que la clase real de la instancia era predicha por XGBOOSTING como si tratara de la tercera o cuarta en orden de probabilidad (expresado en tanto por ciento):

1ª	2ª	3ª	4ª	5ª	6ª	7ª	8ª	9ª
81.45	13.25	3.54	0.93	0.42	0.14	0.12	0.09	0.02

Así, tal y como estaba definida nuestra métrica, a cada clase real a la que adjudicábamos una probabilidad de o en nuestra predicción, cometíamos un error de 34.53. Si modificábamos las probabilidades para la primera y segunda clase que predecíamos como ganadora y el resto las dejábamos a cero, aunque sólo supone apenas un cinco por ciento de instancias mal clasificadas, nuestro error subía directamente sobre 0.6 sin importar cómo de bien hubiéramos clasificado el resto de instancias.

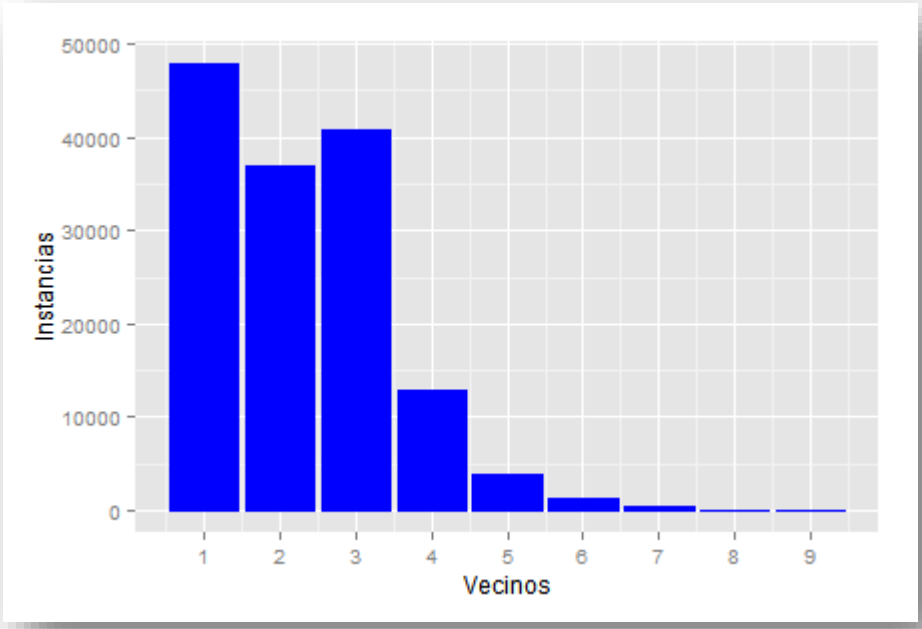
Dynamic One versus One

Se trata de una forma de abordar el problema de la clasificación multiclase distinto al tratamiento pairwise coupling [Hastie98] que utilizan por defecto la mayoría de los algoritmos implementados en R. A partir de la binarización del problema obtenemos por cada instancia una matriz de votación donde cada elemento representa la probabilidad de obtener la clase i frente a la j para cada uno de los $n * (n-1)/2$ clasificadores que tenemos que construir. Es decir, que para nuestro problema de 9 clases, cada matriz de votación para cada una de las instancias que componen la predicción se obtiene de realizar 36 modelos (la probabilidad que obtenemos en el clasificador de la clase uno frente a la dos es la recíproca de hacer la clase dos frente a la uno).

En vez de usar las probabilidades de los 36 clasificadores en nuestra matriz de votación, sólo usaremos las de aquellas clases que sean competentes; esto es, que sean vecinos de la instancia a clasificar. Así, si una instancia está rodeada únicamente por tres clases distintas, sólo usaremos estas tres para generar la matriz de votación.

El número de vecinos que se usarán en este caso será 27 y el clasificador que utilizaremos será Random Forest. Como con éste método sólo podemos obtener la clase ganadora, hay que imponer a mano las probabilidades. Pero gracias a que en la matriz votación podemos obtener una suma de probabilidades para cada clase, en cierta forma es posible establecer su orden. De esta forma podremos identificar la clase más probable y las sucesivas en orden descendente, para así asignarle una probabilidad a mano según corresponda.

Llegados a este punto nos pareció interesante estudiar cómo estaban rodeadas las distintas instancias, si por una, dos o más clases. En la siguiente gráfica se puede ver que hay un número considerable de ellas que están rodeadas por tres o más clases, por lo que al identificar esta situación quizás sería más apropiado ser más conservador y no imponer a la clase ganadora una probabilidad próxima a uno.



El mejor resultado que obtuvimos por esta vía en Kaggle fue 0.71706, usando las condiciones que mostramos a continuación si el número de clases distintas que rodeaban a la instancia era igual a 1, a 2 o mayor que dos. En cierta manera, fijarnos en cómo está rodeada una instancia nos permite establecer un segundo criterio de certeza de cómo de buena ha sido la clasificación del OVO:

Orden	1	2	>2
1	0.999	0.995	0.815
2	0.0003	0.001	0.132
3	0.0002	0.0005	0.036
4	0.0001	0.0001	0.0093
5	0.0001	0.0001	0.0042
6	0.0001	0.0001	0.0014
7	0.0001	0.0001	0.0012
8	0.00005	0.00005	0.00085
9	0.00005	0.00005	0.00005

Dynamic One vs. One y ensemble.

En la recta final de la competición se decide usar la información que proporciona Dynamic One vs. One con Random Forest (OVO) para intentar mejorar de alguna manera las probabilidades que obteníamos con el ensemble que se ha descrito antes.

Para ello se extraía la siguiente información del OVO:

- Suma de las probabilidades de la matriz votación para cada clase: es el resultado de sumar las filas de la matriz de votación dinámica, lo que nos puede dar en cierta forma una medida de probabilidad para cada una de las clases.
- Orden de las clases: a partir de la suma anterior se obtiene el orden de probabilidad de la clase a la que podría pertenecer una instancia. A diferencia del apartado anterior, los elementos que faltan en la dinámica se completan con la estática.
- Distancia de los vecinos más cercanos y clase: esto se obtenía del knn que teníamos que usar para reconstruir la matriz de votación.

Con estos datos se plantean dos estrategias diferentes:

Estrategia 1

A partir del conjunto entrenamiento usamos el veinte por ciento de los datos para crear una partición de test. Usando este particionado creamos un ensemble como el ya descrito y un OVO, de tal manera que sabemos las clases verdaderas que hemos destinado para el conjunto test y las probabilidades que predice el ensemble.

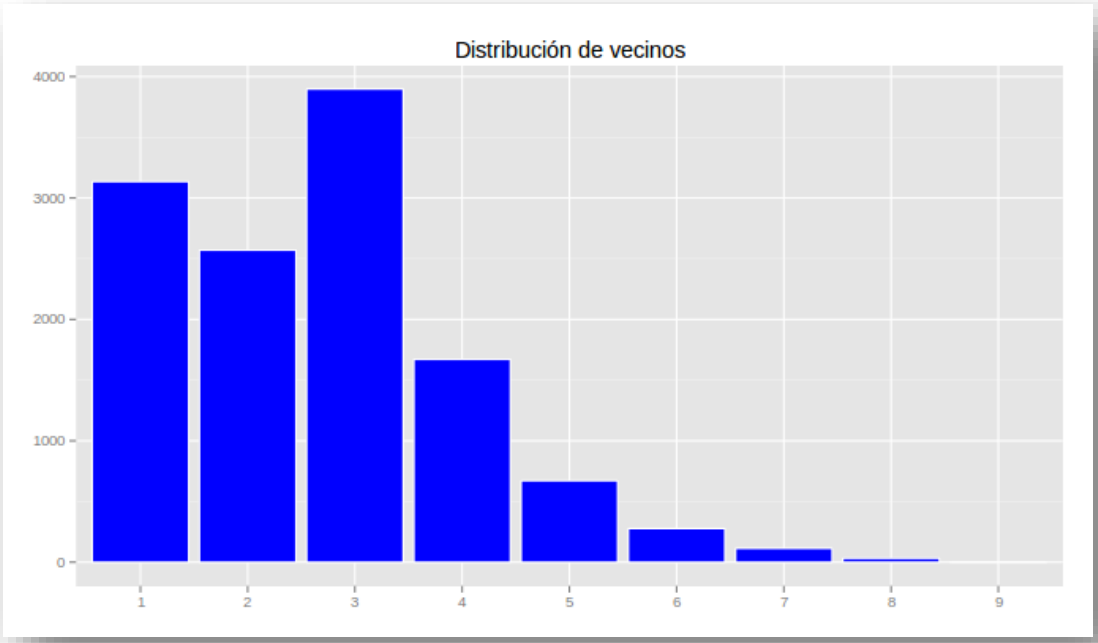
Se trata de generar reglas adhoc usando la información del OVO para cada clase predicha por el ensemble, de tal manera que gracias a ellas podamos estar seguros que el conjunto de instancias predichas que cumplen estas reglas coinciden en clase con la partición del test. Por tanto, podemos modificar sus probabilidades al alza porque en teoría nos estaríamos asegurando por otra vía que la clase predicha coincide con la real.

Sin embargo, y antes de comentar esta estrategia con más profundidad, el método plantea un par de inconvenientes. El primero es que el número de instancias que pueden cumplir las condiciones no es muy alto y por lo general las probabilidades que posee es próximo a uno. El segundo es que se necesitaría generar un conjunto muy elevado de instancias predichas y que nos den la suficiente variabilidad para que al extrapolar las condiciones de nuestro conjunto de prueba al de test (de la competición) se puedan seguir cumpliendo a rajatabla. Cuando digo a rajatabla me refiero a que si esa condición nos aseguraba que sólo la cumplía un subconjunto pequeño de datos de la clase 3, cuando lo extrapolo al conjunto test de la competición quiero no lo cumpla ninguna otra instancia que realmente pueda pertenecer a otra clase. Por tanto, no me queda más remedio que rezar porque que los datos en el conjunto test no hagan cosas raras. En nuestro caso partíamos de solamente 12371 instancias con las que poder obtener estas reglas, por lo que era muy difícil que esta estrategia diera resultado.

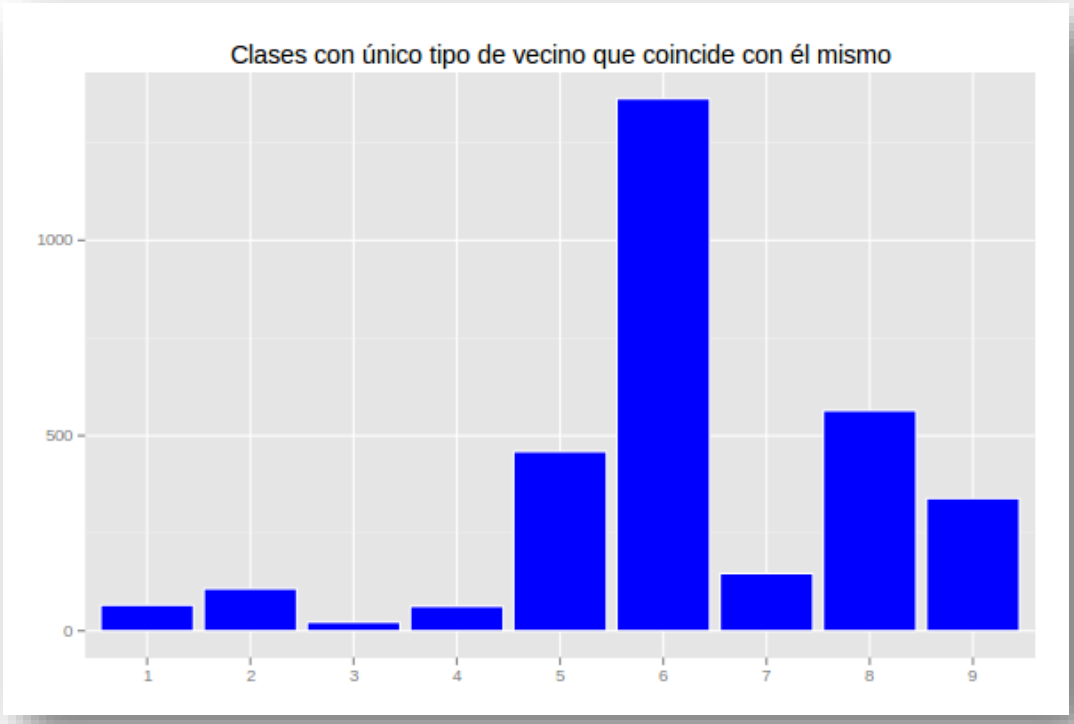
Lo primero que calculamos fue el número de instancias en que OVO y el ensemble coincidían en la primera clase, que fue de 10732 de las 12371 instancias que teníamos de partida. Es decir, que coincidían en un 86.7 por ciento de los casos. Sabíamos que el

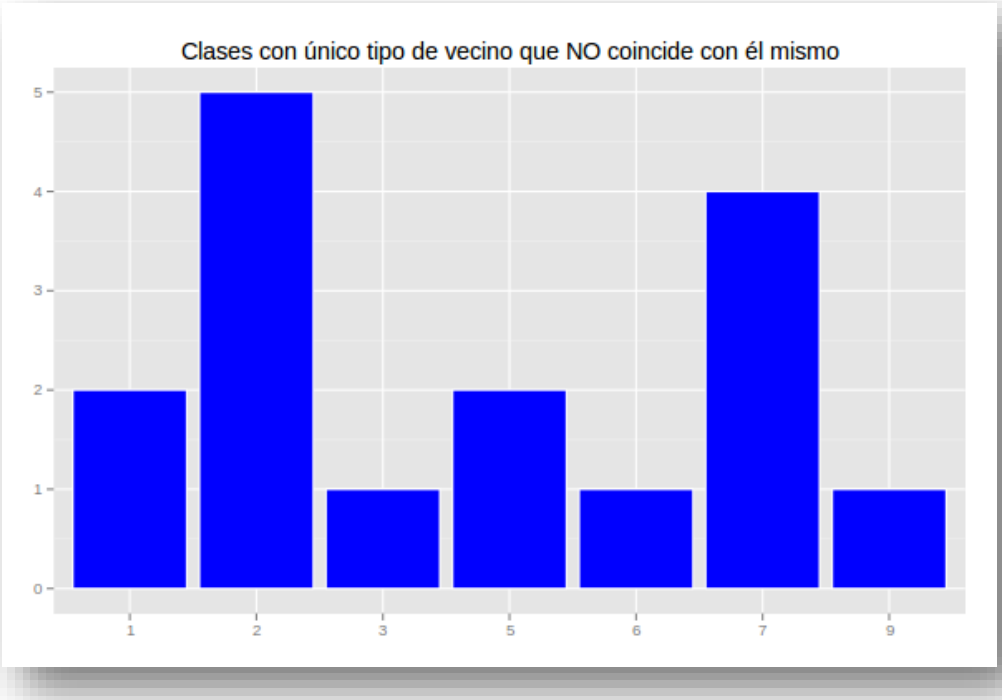
ensemble tenía una tasa de acierto del 83 por ciento, después de haber subido un resultado en el que marcábamos con uno aquel que el ensemble lo daba como clase ganadora y cero el resto.

La distribución de vecinos, entendiendo como el número de clases distintas que conforman los 27 vecinos que rodean a cada una de las instancias de la partición test, es como sigue:

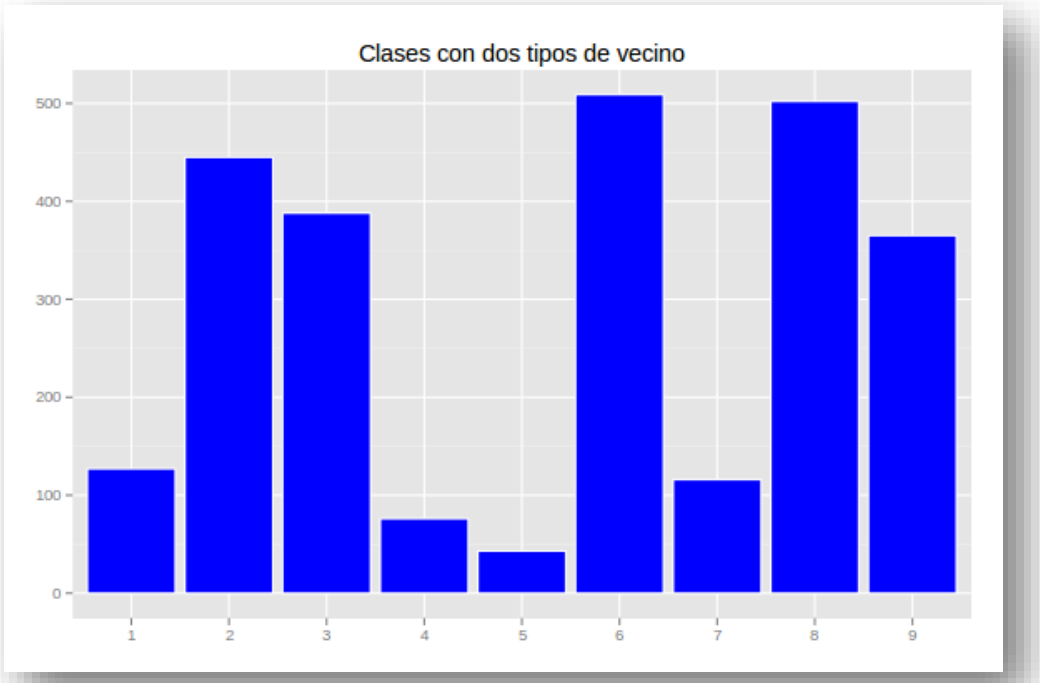


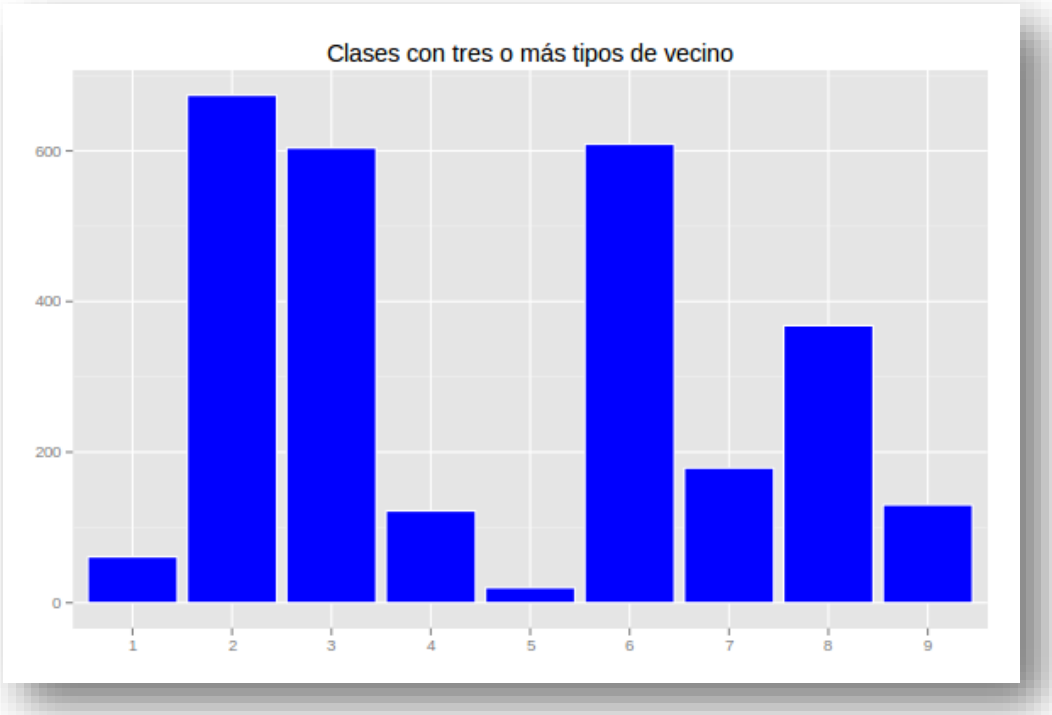
Por lo que seguimos teniendo muchas clases ruidosas y no nos podemos fiar únicamente en este dato como condición. Se puede en las siguientes gráficas cuáles de ellas presentarán por tanto más dificultades:





Se da el caso que existen instancias cuya clase no coincide con ninguno de los vecinos que la rodean, aunque se da en pocos casos.





La clase 2 y 3 están muy mezcladas con instancias de otras clases mientras que la seis tiene muchas instancias rodeadas exclusivamente por ella y también muchas otras que están muy mezcladas.

Las condiciones que utilizamos son las siguientes, siempre partiendo de una condición indispensable (y obvia) como es que la clase del test, la que daba ganadora el OVO y la que daba ganadora el ensemble tenían que coincidir:

- Que la probabilidad de la instancia que se obtenía a partir de la matriz de votación estuviera por encima de un cierto valor.
- Que los vecinos que la rodeara solo tuvieran un número determinado de clases distintas.
- Que la condición anterior se pudiera acotar a un número distinto de vecinos.
- Que además de las dos anteriores, esos vecinos no estuvieran lejos de la instancia (entendido como tal que su distancia sea menor que la distancia media de todos los vecinos más la desviación típica).

Primero se aplicaban en nuestros conjuntos y cuando encontrábamos reglas válidas para una clase en concreto y un número de determinado de clases distintas que rodearan a la instancia que variaba entre uno y tres, las extrapolábamos al conjunto de test de la competición.

Partiendo de las 12371 instancias de prueba, este método modificaba 1128 con el conjunto de reglas que establecimos, pudiendo pasar de **0.4965** que partíamos a un **0.4942**.

Aplicándolas al conjunto test compuesto por 144368 instancias, conseguíamos modificar 12070, pasando en Kaggle de **0.41354** a **0.44580** ☹.

Estrategia 2

En esta segunda forma de procesar la salida se utilizó un cambio mediante una técnica de ensamble, utilizando las predicciones de los modelos obtenidos anteriormente. Esta idea se basaba en utilizar varias soluciones. Después comentaré exactamente los métodos que se utilizaron para la experimentación.

Para mejorar nuestro resultado, se intentó restar probabilidad a las clases con menor probabilidad del mejor método (en nuestro caso el ensamble comentado en la siguiente sección). Las cuestiones que nos planteamos eran ¿cuánto restarle a cada probabilidad? Y ¿qué heurística utilizar? ¿Cuántas probabilidades cambias para minimizar el error introducido?

Para resolver esto se ideó el siguiente método:

1. Solo se cambiaban las 2-3 probabilidades menores de cada instancia de la solución (dependiendo si coincidían las dos o tres últimas). Dos heurísticas posibles:
    - Coincidían en el mismo orden
    - Coincidían en las tres últimas clases
2. Se realizaba el cambio cuando estas probabilidades no superaban dos heurísticas:
    - Su suma >0.3 (si eran las 3 últimas), 0.2 (si eran dos).
    - Que alguna de las probabilidades fuera mayor a un umbral (nosotros utilizamos 0.15)

3. Por último, para restar la probabilidad se utilizó un umbral mediante el cual restábamos un porcentaje a cada probabilidad. Aunque se podían utilizar diferentes configuraciones la nuestra fue:
    - Restar a la última un 60% a la penúltima 40% y a la penúltima un 20%. Esta probabilidad se sumaba y se repartía entre las dos mejores probabilidades (70% a la primera y 30% a la segunda)

Para nuestra prueba en kaggle utilizamos un Random Forest, Xgboost, dos redes neuronales, deeplearning y OVO. EL resultado no fue bueno pues pasamos de 0.41 a 0.43 usando este método. Si hubiéramos tenido más tiempo se podrían haber mejorado los umbrales para minimizar el error introducido y así poder mejorar la solución.

Solución Final

Mejor resultado

Para la solución final se realizó un ensamble de los mejores métodos , pues estudiando las probabilidades que daban cada uno de ellos, se podía comprobar por ejemplo que las redes neuronales eran capaces de arriesgar más con probabilidades altas y funcionaban mucho mejor en las clases minoritarias, mientras que el boosting era más conservador en las instancias difíciles. Por ello, al unir estas dos soluciones espere una mejora en el resultado de la predicción.

Nuestro ensamble tenía la siguiente estructura:

Con esta solución, en Kaggle se obtuvo un resultado de 0.412 (en la parte pública) y cuando se liberó la parte privada en el cierre de la competición nos dio 0.413 obteniendo la posición 77 de 3590.

Repositorio github: https://github.com/BaCaIs-etho/Kaggle_Otto_Group

Perfil de kaggle:

- https://www.kaggle.com/cjferba
- https://www.kaggle.com/ismael

Bibliografía

[1] Kaggle, «Kaggle Otto group product classification challenge,» 17 3 2015. [En]. Available: https://www.kaggle.com/c/otto-group-product-classification-enge. [Último acceso: 2015 5 30].

[2] Cambridge University Press, «Tf-idf weighting,» 07 04 2009. [En línea]. lable: http://nlp.stanford.edu/IR-book/html/htmledition/tf-idf-weighting-1.html. mo acceso: 16 4 2015].

[3] Wikipedia, «Anscombe transform,» 14 9 2014. [En línea]. Available: //en.wikipedia.org/wiki/Anscombe_transform. [Último acceso: 26 4 2015].

[4] N. V. B. K. W. H. L. O. &. K. W. P. Chawla, «SMOTE: synthetic minority over-›ling technique,» *Journal of artificial intelligence research,* nº 16, pp. 321-357, 2002.

[5] K. A. R. e. a. Bee Wah Yap, «An Application of Oversampling, Undersampling, ing and Boosting in Handling Imbalanced Datasets,» de *Advanced data and mation engineering*, 2014.

[6] Kaggle, «Kaggle Otto Group evaluation,» 17 4 2015. [En línea]. Available: s://www.kaggle.com/c/otto-group-product-classification-enge/details/evaluation. [Último acceso: 10 5 2015].

[7] B. Ripley, «nnet,» CRAN, 2015.

[8] C. Bergmeir y J. M. Benítez, «RSNNS,» 19 2 2015. [En línea]. Available: //cran.r-project.org/web/packages/RSNNS/RSNNS.pdf. [Último acceso: 12 4 2015].

[9] H2O, «Algorithms Deep Learning,» 18 2 2015. [En línea]. Available: http://h2o-se.s3.amazonaws.com/h2o/rel-shannon/12/docs-website/h2o-/index.html#Data%20Science%20Algorithms-Deep%20Learning. [Último acceso: 2015].

[10] Sander Dieleman and contributors, «Documentation,» 2014. [En línea]. lable: http://lasagne.readthedocs.org/en/latest/user/layers.html. [Último acceso: 1 5].