UNIVERSIDAD DE GRANADA

APS Failure at Scania Trucks Data Set

Práctica Final de Aprendizaje Automático

Autores:

Alejandro Alonso Membrilla

Adolfo Soto Werner

Junio 2021

Contents

1 Introducción y Descripción del Problema

Este proyecto consiste en un análisis completo de un problema de Machine Learning seleccionado del repositorio oficial de la *University of California, Irvine*. El dataset que vamos a analizar es *APS Failure Scania Trucks Data Set*, para solucionar el problema de clasificación binaria que este supone mediante modelos de Aprendizaje Automático, tanto lineales como no lineales.

Este conjunto de datos se presentó en el *Industrial Challenge 2016 at The 15th International Symposium on Intelligent Data Analysis (IDA16)*, y consiste en una serie de medidas de uso diario recolectadas de camiones de la marca *Scania*. El número total de elementos es 60000 cada uno de ellos con 170 características diferentes. Un conjunto adicional con 16000 ejemplos (aproximadamente un 21% de la suma total) se aporta a modo de conjunto de test.

Aunque las variables medidas para cada elemento son anónimas por petición del fabricante, según la información adjunta al dataset contamos con dos tipos de variables, aquellas que son contadores simples y otras que en conjunto representan histogramas. Todas las variables representan valores enteros pero serán tratadas como números flotantes.

La clasificación se centra en el Sistema de Presión de Aire (APS), que genera aire presurizado utilizado para varias funciones del camión como son el sistema de frenos y los cambios de marcha. Esta separación discierne entre dos clases distintas, aquellos casos en los que se han dado fallos en componentes que están relacionados con este sistema y aquellos en los que los fallos se han dado en componentes completamente independientes del mismo.

A partir de la descripción anterior, describimos formalmente el problema de aprendizaje como la búsqueda de una función $f : \mathbb{R}^{170} \to \{neg, pos\}$ que asigne una categoría (neg, componentes no relacionadas, o pos, componentes relacionadas) a un dato conformado por un vector de 170 números reales, y que sea capaz de predecir de la forma más fiable posible cuándo unas medidas corresponden a un error en componentes relacionadas o no relacionadas con el APS. Para esto, aplicaremos métodos y herramientas de aprendizaje supervisado que se ajusten y busquen generalizar los datos etiquetados provistos por el conjunto de datos utilizado.

El código implementado para los experimentos ha sido escrito en Python, haciendo uso de la biblioteca *Numpy* para álgebra lineal y generación de números pseudo-aleatorios, *Scikit-Learn* para técnicas y herramientas de estadística y aprendizaje automático, y *Matplotlib* y *Seaborn* para la visualización de los resultados. En lo que respecta a la ejecución de algoritmos que hacen uso de una semilla, para garantizar la replicabilidad de nuestros resultados dicha semilla se ha fijado a un valor al principio del código implementado, se utiliza en todos los algoritmos pseudo-aleatorios y no se cambia en ningún momento a lo largo de la ejecución.

2 Análisis Exploratorio y Preprocesamiento

En esta sección describiremos todo el proceso realizado con los datos previo al entrenamiento y selección de modelos, comenzando por la carga de los conjuntos de entrenamiento y test. Es conveniente que nuestros datos de entrenamiento se encuentren debidamente barajados, o podríamos encontrar el problema de no representar correctamente el conjunto de entrenamiento más adelante al extraer subconjuntos para validación. Por ello barajamos los datos usando una semilla aleatoria fijada previamente para todo el experimento.

Será necesario realizar transformaciones sobre el conjunto de entrenamiento para favorecer el aprendizaje (tratamiento de valores perdidos, normalización...). Cabe destacar que algunas de las transformaciones hechas sobre el conjunto de entrenamiento tendrán que replicarse sobre el conjunto de test para que tenga sentido aplicar sobre él los modelos entrenados. Siempre que esto suceda se indicará explícitamente.

En lo que respecta a la **codificación de las variables**, como nuestro dataset no cuenta con variables categóricas no ha sido necesario codificarlas.

Por tanto, empezaremos tratando los **valores perdidos**, abundantes en nuestro conjunto de entrenamiento. La figura 1 muestra que porcentaje de valores perdidos hay en las columnas (características) del conjunto, ordenándolas en orden decreciente de valores perdidos. Hay 8 variables con más de un **50%** de valores perdidos. Esto quiere decir que más de la mitad de la información de estas características debe ser inferida a partir del resto de la muestra para usarse en el entrenamiento. Por tanto, para no introducir un sesgo al manipular estas variables sin una información experta de su contenido, decidimos prescindir de las mismas (las eliminamos del conjunto de entrenamiento y del de test).

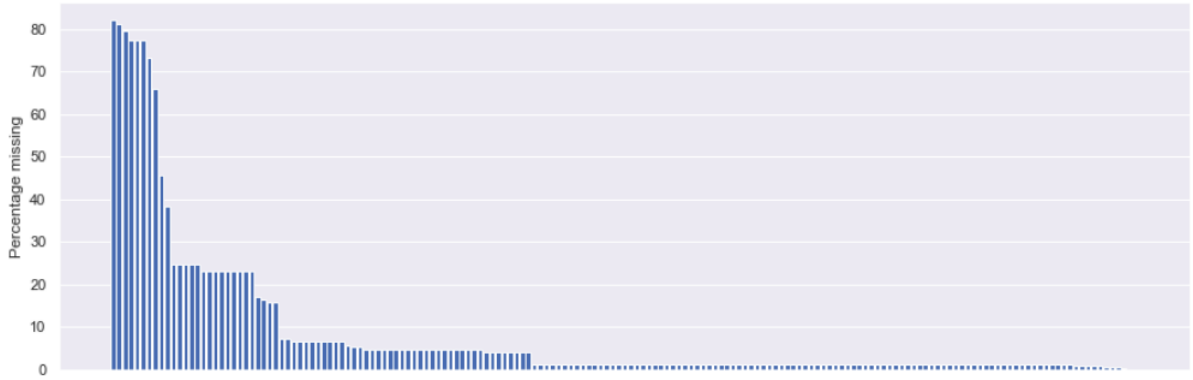


Figure 1: Porcentaje de valores perdidos por columnas.

También hemos eliminado todas las filas (ejemplos) que tienen más de un 20% de valores perdidos, siguiendo las indicaciones dadas en el guión de la práctica. Aquellos datos que mantengan valores perdidos después de la criba anterior, se tratarán de la forma indicada en el guión de prácticas para variables numéricas: asignándole un valor aleatorio en el intervalo $[\mu - 1.5\sigma, \mu + 1.5\sigma]$, donde μ y σ son la media y desviación típica, respectivamente, de la variable a la que pertenece el valor perdido (naturalmente, solo se tienen en cuenta los valores no perdidos a la hora de calcular estos estadísticos). Aplicamos este mismo método para sustituir los valores perdidos en test, utilizando la media y desviación típica del conjunto de entrenamiento para generar los datos aleatorios.

Continuaremos analizando **cuan balanceadas** se encuentran las dos clases en las que se divide nuestro dataset. En la figura 2 vemos la distribución de ambas en el conjunto de entrenamiento, y ambas se encuentran claramente desbalanceadas. Concretamente, después de eliminar las filas con muchos valores perdidos, contamos con **56070** ejemplos de la clase *neg*, y **616** de la *pos*.

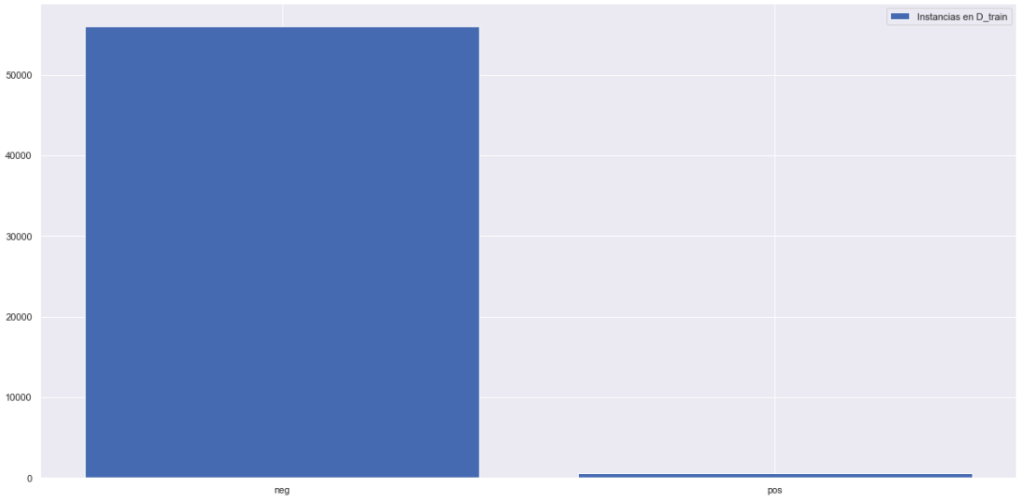


Figure 2: Número de ejemplos de entrenamiento de cada clase (*neg* a la izquierda y *pos* a la derecha).

Vamos a analizar si es necesario **normalizar** nuestro conjunto de datos. Diferencias notables en escala entre variables pueden suponer un problema para muchos métodos en Aprendizaje Automático. En este artículo se indica que algoritmos que "ajustan un modelo a partir de sumas ponderadas de los *inputs*",

como Regresión Logística o MLPs, o que "usan medidas de distancias", como SVMs, se ven afectados por la normalización. Esto es especialmente cierto cuando se desea usar algún tipo de regularización sobre los pesos, puesto que escalas muy distintas en los datos implican diferencias artificiales en los pesos que dificultan la optimización con restricciones sobre dichos pesos y provocan inestabilidad (*The Elements of Statistical Learning* cap. 3.4.1).

Visualizamos las **medias y desviaciones típicas** de las distintas variables calculadas para el tratamiento de valores perdidos (figura 3). Remarcamos que los gráficos de barras mostrados se encuentran en escala logarítmica. Las diferencias en magnitud entre las distintas variables son demasiado grandes, justificando normalizar el conjunto de datos. Aplicaremos una **estandarización**: restar cada atributo por la media de su variable correspondiente y dividirlo entre su desviación típica. La distribución resultante de cada variable puede verse en el diagrama de cajas de la figura 4.

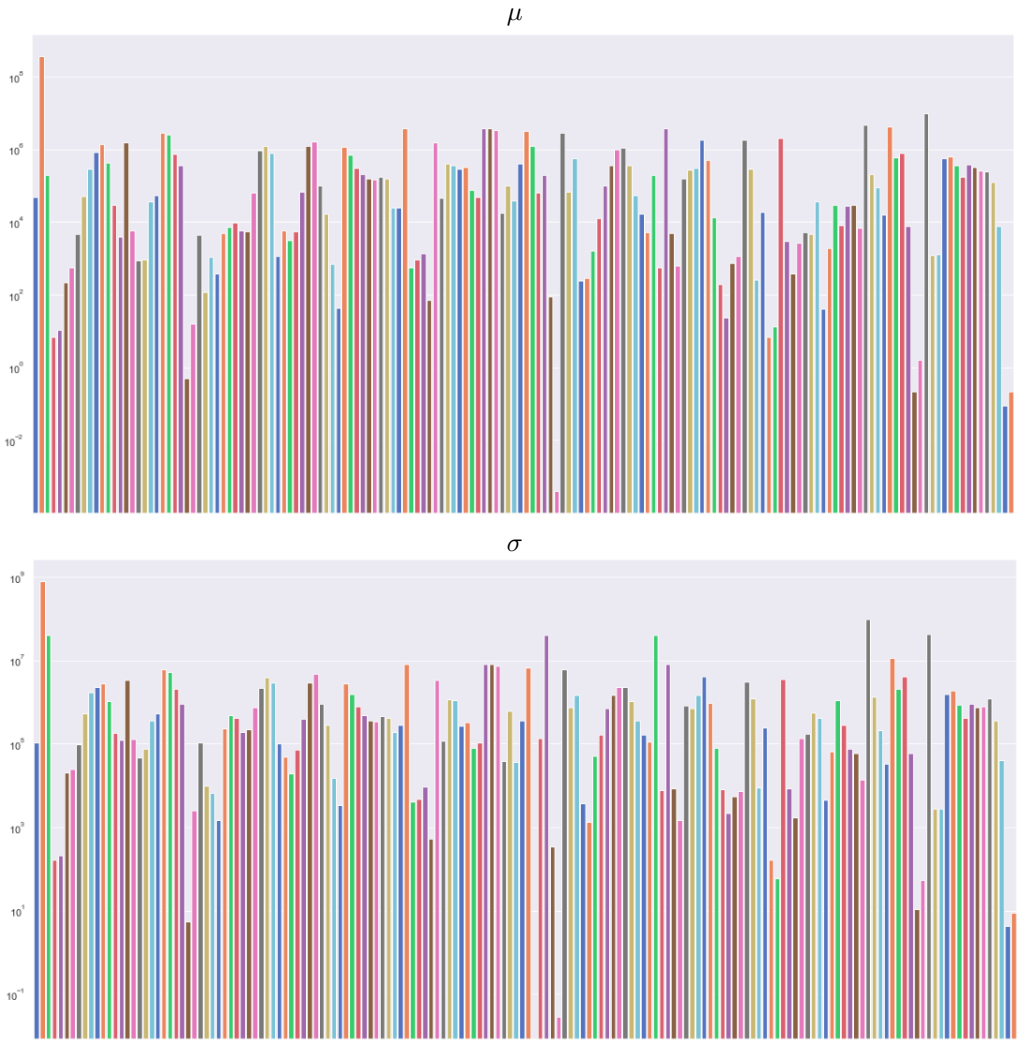


Figure 3: Medias (arriba) y desviaciones típicas (abajo)

La transformación aplicada sobre el conjunto de entrenamiento debe replicarse sobre el de test. Esto no quiere decir restarle al conjunto de test su media y dividir entre su desviación típica, sino restarle la media del conjunto de entrenamiento y dividirlo entre la desviación típica de este mismo conjunto. En resumen, es necesario **aplicar al conjunto de test exactamente las mismas transformaciones** que aplicamos sobre el de entrenamiento.

La figura 4 también podría sernos útil para decidir sobre el tratamiento de **valores extremos**. En este caso, al no contar con información de la naturaleza de nuestros atributos y al no apreciar visualmente *outliers* flagrantes, decidimos no eliminar valores extremos.

Por último, estudiaremos la redundancia de información en nuestro conjunto de datos. La presencia

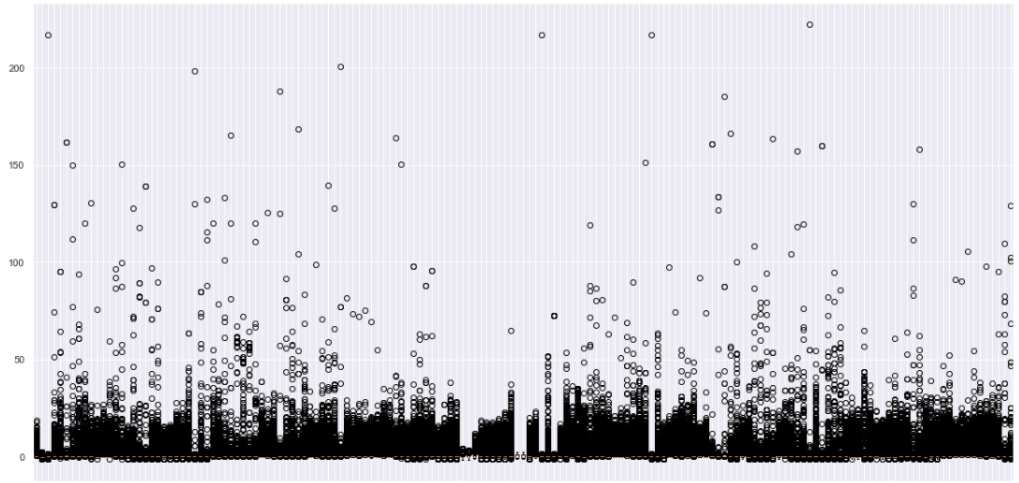


Figure 4: Diagrama de cajas de cada variable.

de variables altamente correladas puede ocasionar problemas a la hora de entrenar nuestro modelo, y por tanto es conveniente estudiar si este es el caso y eliminar dichas dependencias. La figura 5 muestra los coeficientes de correlación de Pearson para cada dos variables del conjunto de entrenamiento. Vemos que existen grupos de variables media e incluso altamente correladas, lo cual justifica el uso de técnicas de reducción de la dimensionalidad.

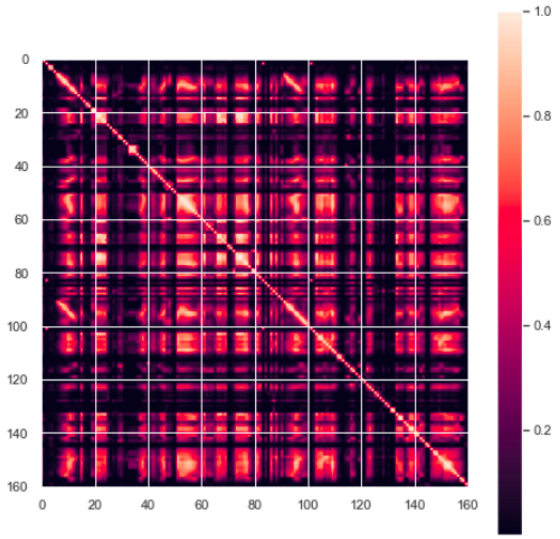


Figure 5: Matriz de correlaciones entre las 162 variables restantes del dataset.

En este experimento **extraeremos** un subconjunto de las *características más relevantes* aplicando *PCA*, o ***Análisis de Componentes Principales***, que consiste en realizar un cambio de base sobre el espacio en que representamos nuestros datos de forma que las nuevas variables sean las proyecciones sobre esa nueva base, que todas ellas sean incorreladas entre sí, y que se encuentren ordenadas de acuerdo al porcentaje total de la varianza (entendida como medida de información) que expliquen de los datos. Decidimos quedarnos con las primeras componentes principales que expliquen un 98% de la varianza del dataset, lo cual nos deja con 100 características (hemos eliminado 62 dimensiones de escasa importancia). En la figura 6 vemos el porcentaje de varianza explicada por cada una de ellas. La figura 7 muestra las correlaciones entre las componentes principales y, tal como se esperaba, estas son mutuamente incorreladas.

Nos gustaría tener una forma de medir cuanta información parece aportar cada variable para resolver nuestro problema. Al contar con un dataset con un ratio instancias/atributos relativamente alto, es posible aplicar métodos no paramétricos para estimar la dependencia entre las etiquetas y cada uno de

Figure 6: Porcentaje de varianza explicada por cada una de las 100 primeras componentes principales.

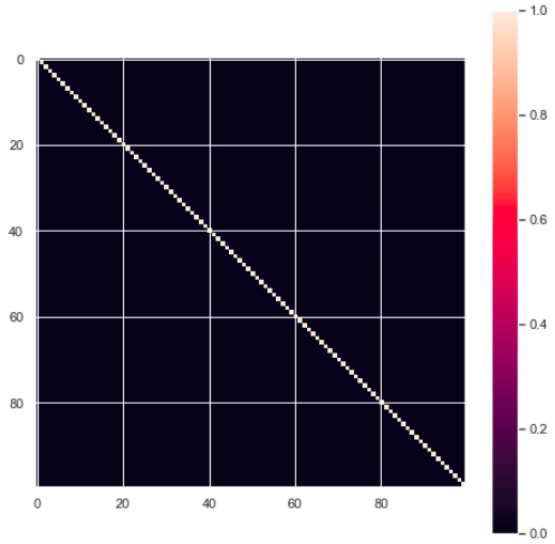


Figure 7: Correlaciones de las 100 primeras componentes principales.

los atributos. Esto es lo próximo que vamos a probar, con la función *mutual_info_classif()* de Scikit-Learn. Para variables continuas, esta función agrupa los distintos valores observados usando un esquema de K-NN, discretizando el espacio, y calculando **una medida de información mutua** para variables discretas tal como se puede ver en este artículo. Al considerar la información mutua de cada variable por separado, el método puede ser invariante a centro y escala y no es necesario normalizar los datos previamente.

El resultado de aplicar *mutual_info_classif()* sobre cada variable es un número real entre el 0 y el 1, que es más alto cuanta mayor dependencia haya entre dicha variable y el vector de etiquetas. El número de vecinos utilizado para el agrupamiento mediante el vecino más cercano ha sido fijado en 3 vecinos, justificándonos en los resultados expuestos en teoría que abalan este valor empíricamente (tema 9, pág. 16). La gráfica en la figura 8 representa los resultados. Todas las variables restantes aparentan tener una dependencia relativamente fuerte con el vector de etiquetas, por lo que decidimos no eliminar ninguna.

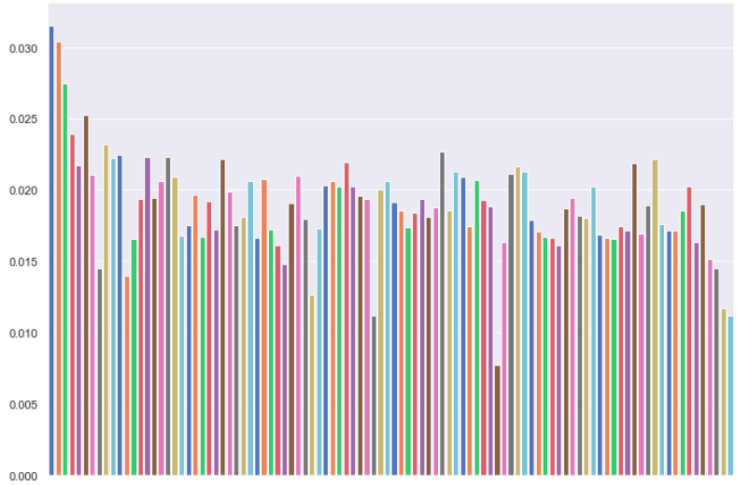


Figure 8: Información mutua de cada variable con el vector de etiquetas.

3 Métricas de Evaluación

Cada modelo empleado utilizará una función de pérdida distinta, puesto que parten de diferentes formulaciones matemáticas del problema de clasificación. Por tanto, estas son las funciones que hemos utilizado para seleccionar la mejor solución para cada combinación de hiperparámetros, y las explicaremos más adelante al tratar su modelo correspondiente. Sin embargo, para seleccionar un mejor modelo de entre todos con los que hemos experimentado utilizamos una métrica distinta, que aplica una penalización distinta a cada tipo de error cometido.

Esta métrica de evaluación se basa en la suma ponderada de los falsos positivos (o datos de la clase negativa etiquetados como positivos) y falsos negativos (o datos de la clase positiva etiquetados como negativos) ambos ponderados. Los pesos aplicados a ambos valores han sido 500 para los falsos negativos y 10 para los falsos positivos, siguiendo el procedimiento establecido por los autores de la base de datos y las normas del IDA16. Con esta ponderación apreciamos que se da mucha mas importancia a un falso positivo que a un falso negativo ya que, como hemos mencionado previamente, las dos clases de la base de datos están muy desbalanceadas. Por tanto, la función que queremos minimizar es la siguiente:

$$SCORE(tp, fp, fn, tn) = 10fp + 500fn$$

donde tp, fp, fn y tn representan, respectivamente, los verdaderos *pos*, falsos *pos*, falsos *neg* y verdaderos *neg*.

Además de esta métrica, para elegir nuestro modelo recurrimos a las matrices de confusión en las que vemos la cantidad de predicciones presentes en las 4 categorías anteriores, que representaremos con el aspecto siguiente:

$$\begin{pmatrix} True\ neg & False\ pos \\ False\ neg & True\ pos \end{pmatrix}$$

También calcularemos la *precisión* y el *recall*. Esto nos ayudará a apreciar visual y cuantitativamente las consecuencias del desbalanceo y a seleccionar los mejores modelos en presencia de la paradoja de la exactitud.

4 Selección de Modelos

El objetivo de este proyecto final es comparar los resultados ofrecidos por un modelo lineal frente a aquellos ofrecidos por una serie de modelos no lineales. Dados los datos de entrenamiento, cada modelo se entrenará 5 veces con cada posible combinación de parámetros (k-fold cross-validation con $k = 5$), cada vez eligiendo un subconjunto con una quinta parte de los datos que hará de conjunto de validación, para testear el modelo resultante de entrenar con los parámetros establecidos y el resto de datos de entrenamiento. Durante el proceso de validación cruzada, cada elemento de entrenamiento es usado para validar exactamente una vez. Esto hace que cada dato tenga una única predicción asociada. Estas predicciones pueden compararse con las etiquetas reales para calcular el SCORE asociado, que será nuestra principal métrica de error en el problema de clasificación tal como se ha explicado en la sección anterior.

La **elección del valor de** k se ha realizado teniendo lo siguiente en consideración. Un número de *folds* muy alto reduce el sesgo a la hora de estimar E_{out} con respecto a uno bajo. El criterio de *Leave One Out* (Deja Uno Fuera) dará aproximadamente una estimación insesgada del error de test, como se ha visto en teoría. Sin embargo, este método posee una varianza muy elevada y, principalmente, es computacionalmente inviable para conjuntos de datos grandes. Elegir un número de *folds* más reducido busca un ajuste entre el sesgo que esto introduce, la varianza de la estimación y el ahorro computacional.

De acuerdo con An Introduction to Statistical Learning, cap. 5.1.4, los valores de k más utilizados son $k = 5$ y $k = 10$, puesto que "se ha demostrado empíricamente que producen estimaciones del error de test que no sufren excesivamente de un sesgo ni de una varianza demasiado altos". Nos fundamentamos en esto para tomar $k = 5$, como se ha dicho anteriormente.

Probaremos cada tipo de modelo comparando sus resultados para una serie de hiperparámetros, debidamente justificados, mediante una estrategia tipo *grid search*, esto es, probando las distintas combinaciones de hiperparámetros y eligiendo aquella que logre un mejor SCORE en validación cruzada.

4.1 Modelos Analizados

El primer resultado que vamos a comentar es un modelo lineal sencillo que ignora el desbalanceo de los datos y que usaremos como referencia para ver como nuestros modelos mejoran al tenerlo en cuenta.

4.1.1 Regresión Logística

El primer modelo que vamos a pasar a analizar es **regresión logística**. Tanto RL como las máquinas de soporte vectorial son modelos lineales para clasificación de uso extendido en Aprendizaje Automático. Ambos admiten regularización y *Scikit-Learn* permite entrenarlos con diversos métodos de optimización. Sin embargo, según lo estudiado en teoría, regresión logística funciona mejor que el SVM con kernel lineal en casos con ruido, y en eso nos justificamos para elegir este como nuestro modelo lineal a utilizar.

Como haremos en todos los casos posteriores vamos a fijar los valores de una serie de parámetros y a realizar una búsqueda entre distintos valores de otros. Pasamos a ver los parámetros que hemos dejado fijos:

- *dual = False*

  Este parámetro sirve para especificar que queremos utilizar la formulación primal, ya que la dual solamente tiene sentido cuando tenemos un *dataset* con más características que datos (que no es nuestro caso).

- *max_iter = 1000*

  Es el máximo número de iteraciones que le permitimos realizar al algoritmo de optimización. Lo fijamos a 1000 por limitaciones computacionales.

- *tol = 0.0001*

  Representa la tolerancia que consideramos como criterio de parada. Dejamos el valor por defecto 0.0001, ya que disminuirlo podría aumentar notablemente el tiempo de cómputo.

- *fit_ intercept=True*

  Con este parámetro decidimos si añadir o no un término de sesgo. En nuestro caso lo activamos ya que no se lo hemos incluido manualmente.

- *intercept_ scaling=1*

  Este parámetro se utiliza para fijar el valor base del sesgo. No tenemos motivos para eliminarlo así que lo fijamos a 1 para no influir en el proceso de ajuste del sesgo.

- *class_ weight=None*

  Asigna un peso a cada clase. Sirve como mecanismo contra el desbalanceo. Como en este caso queremos comprobar que sucede si no tenemos en cuenta el balanceo, dejamos el valor por defecto. En el siguiente caso, en el que usamos un clasificador tipo bagging con submuestreo balanceado, dejaremos este valor por defecto al aplicar RL sobre subconjuntos balanceados.

- *multi_ class='auto'*

  Es el método que se utiliza parar resolver problemas multiclase. Al dejar el valor por defecto 'auto' esta decisión se toma internamente y se selecciona el método *one-vs-rest* por ser el dataset de clasificación binaria.

- *warm_ start=False*

  Este parámetro indica que toma como pesos inciales los resultados de la ejecución anterior. Como queremos aplicar validación cruzada, esto sería absurdo, puesto que estaríamos usando como pesos iniciales aquellos pesos aprendidos a partir de pesos que ahora queremos predecir (caeríamos en el data snooping). Es por esto que lo dejamos a False.

- *l1_ ratio = None*

  Este parámetro indica el peso de la regularización l_1 al utilizar el método de regularización *elastic-net* que es una hibridación entre l_1 y l_2. En nuestro caso no tiene sentido su uso ya que no utilizamos este tipo de regularización.

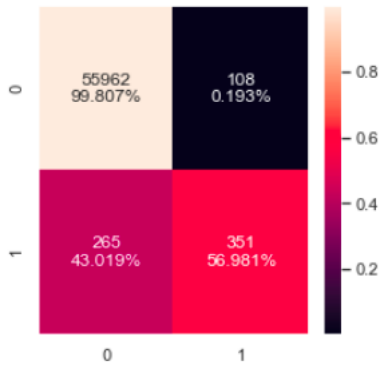
Por otro lado, hemos experimentado con los siguientes parámetros para seleccionar aquellos valores que obtengan mejores resultados.

- $C \in \{1, 10, 100, 1000, 10000\}$

  Este es el inverso del índice de regularización que queremos utilizar en el modelo. Los valores que hemos decidido emplear en la búsqueda son $[1, 10, 100, 1000, 10000]$ ya que el coeficiente de regularización a menudo se fija a 0.1, 0.01,0.001... (ver este artículo y el paper "Regularization Path For Generalized linear Models by Coordinate Descent", Friedman, Hastie & Tibshirani, J Stat Softw, 2010).

- $solver + penalty \in \{saga + l1, lbfgs + l2\}$

  En este caso hemos iterado sobre parejas de algoritmos y tipo de regularización ya que hay algoritmos que no aceptan un determinado tipo de regularización. Hemos incluido en el preprocesado un Análisis de Componentes Principales que nos ha agrupado la mayor parte de la información del dataset en el menor número de componentes posibles. Aunque conservamos hasta el 98% de la varianza explicada, muchas de nuestras variables transformadas contienen muy poca información, y podrían ser inútiles. La regularización *l1* tiende a eliminar variables inútiles, por lo que podría ser interesante utilizarla. El *solver* más eficiente de entre los que *Scikit-Learn* pone a nuestra disposición para entrenar RL con *l1* es *saga*, descrito en este artículo, basado en incremento de gradiente. Por esto mismo, *saga + l1* será nuestra primera opción. Por otro lado, *lbfgs* es un método ampliamente utilizado y muy competitivo en cuanto a resultados y número de iteraciones. Aunque solo permite usar *l2*, consideramos de interés experimentar con este optimizador.



Precision *neg* : 0.99528
Recall *neg* : 0.99807

Precision *pos* : 0.7647
Recall *pos* : 0.5698

Figure 9: Matriz de confusión de Regresión Logística en validación cruzada. Medidas de *precision* y *recall*.

Los resultados obtenidos a lo largo del proceso de búsqueda entre los valores de los parámetros se recogen en la tabla 1, resaltando el mejor de los obtenidos según la métrica que hemos explicado previamente. Vemos que obtenemos sin mucha dificultad una exactitud de más del 99%. En la figura 9 vemos que el modelo no parece clasificar correctamente la clase minoritaria: un 0.5698 de *recall* indica que solo aproximadamente el 57% de las instancias de la clase minoritaria se clasifican correctamente. Esto se debe a la presencia de desbalanceo y a una tendencia de los modelos a ignorar este desbalanceo para minimizar el error de clasificación. El modelo "aprende a ignorar" la clase minoritaria para clasificar la mayoritaria con el mayor *accuracy* posible.

Solver	**Tipo de Regularización**	**C**	**ACCURACY**	**SCORE**
saga	L1	1	0.9934	133580
saga	L1	10	0.9934	133090
saga	L1	100	0.9934	132590
saga	L1	1000	0.9934	132590
saga	**L1**	**10000**	**0.9934**	**132590**
lbfgs	L2	1	0.9934	134060
lbfgs	L2	10	0.9935	133030
lbfgs	L2	100	0.9935	133560
lbfgs	L2	1000	0.9935	133550
lbfgs	L2	10000	0.9935	133550

Table 1: Resultados para regresión logística.

Para lidiar con el problema del desbalanceo hemos utilizado en los experimentos que siguen la técnica de construir *ensembles* con múltiples instancias de cada uno de los modelos en un ***BalancedBaggingClasiffier***, de la biblioteca *ImbLearn*. Así, entrenamos cada modelo tomando subconjuntos balanceados del conjunto de entrenamiento de forma que aprende teniendo en cuenta la clase minoritaria.

Los parámetros utilizados en cada bagging son los siguientes:

- n_estimators=300

  Número de estimadores/modelos base que componen nuestro ensemble.

- max_samples=1500

  Tamaño de la muestra que se toma para entrenar cada estimador. Dado que contamos con algo más de 600 ejemplos de la clase minoritaria, una muestra de 1500 ejemplos debería permitir entrenar los modelos base conservando un balance entre la clase mayoritaria y la minoritaria.

  300 estimadores entrenados a partir de 1500 elementos ($300 \cdot 1500 = 450000$) son suficientes para garantizar que todo el conjunto de entrenamiento intervenga en el aprendizaje.

- max_features=1.0

  Es la proporción de las caracterísitcas totales con la que entrenar cada estimador. Tomamos el total.

- sampling_strategy=1.0

  Indica el ratio aproximado de elementos de la clase mayoritaria con respecto a la minoritaria. No afecta en este caso.

En nuestro caso, los modelos que hemos decidido utilizar como base para el bagging han sido:

4.1.2 Balanced Bagging de Regresión Logística

En este modelo utilizamos la misma técnica que en el caso anterior, pero con la diferencia de que esta vez nuestro modelo sí que tiene en cuenta el balanceo de los datos. Tanto los parámetros que hemos fijado como sobre los que hemos realizado la búsqueda y los valores de los mismo son idénticos al caso anterior.

Como hemos mencionado antes cada modelo tiene su propia función de pérdida, en este caso dicha función, dependiendo del tipo de regularización a usar (l_1 o l_2 respectivamente), la función de pérdida es una de las siguiente:

$$\min_{w,c} ||w||_1 + C \sum_{i=1}^{n} log(e^{-y_i(X_i^T w+c)} + 1)$$

$$\min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} log(e^{-y_i(X_i^T w+c)} + 1)$$

4.1.3 Balanced Bagging de Neural Networks

Para usar modelos basados en redes neuronales hemos recurrido a la bibilioteca *neural_network* de *sklearn*, en concreto hemos utilizado el perceptrón multicapa orientado a clasificación que se corresponde con la clase *MLPClassifier*. La motivación detrás del uso de estos modelos reside en su flexibilidad, tanto en su complejidad como en su regularización. Son de uso muy extendido en Aprendizaje Automático y existe mucha literatura al respecto que puede usarse para respaldar los hiperparámetros de nuestro experimento. Además, al utilizar subconjuntos pequeños del conjunto de entrenamiento, necesitamos reducir el tamaño de la red lo que nos permite entrenarla con métodos numéricos eficientes, como el *lbfgs*.

Como en todos los modelos estudiados en esta práctica hemos fijado los valores de una serie de parámetros y realizado una búsqueda sobre conjuntos de valores de otros diferentes. Comenzamos explicando los que hemos fijado:

- *activation = 'relu'*

  Este parámetro indica la función de activación de las neuronas de las capas ocultas. En nuestro caso el valor del parámetro que hemos dejado es 'relu' lo que implica que la función de activación es $f(x) = \max(0, x)$.

- *solver='lbfgs'*

  Se utiliza para indicar el algoritmo de optimización a utilizar para resolver el problema. Hemos utilizado 'lbfgs' ya que para conjuntos pequeños (que es nuestro caso) se "ha observado que *lbfgs* converge más rápido y con mejores soluciones".

- *max_iter = 1000*

  Con este parámetro indicamos el máximo número de iteraciones del algoritmo. Lo hemos fijado en 1000.

- *tol = 0.0001*

  Este parámetro indica la tolerancia en el proceso de optimización. Cuando la función de pérdida no mejora al menos hasta este valor se considera que la convergencia ha sido alcanzada y se para el entrenamiento. La hemos fijado a 0.0001 en un compromiso entre búsqueda de soluciones óptimas y tiempo de ejecución.

- *warm_start = False*

  Con este parámetro podemos indicar si queremos o no reutilizar la solución de la anterior llamada a *fit* como inicialización. Igual que en RL, no tiene sentido su uso para este caso.

- *max_fun = 15000*

  Este parámetro indica el número máximo de llamadas a la función de pérdida. Se utiliza como criterio de parada para *lbfgs* junto a *max_iter* y *tol*. Lo hemos fijado en 15000.

Pasamos ahora a explicar los parámetros y sus correspondientes valores involucrados en la búsqueda de la mejor combinación:

- *hidden_layer_sizes* $\in \{(5,5),(10,10),(25,25),(50,50)\}$

  El i-ésimo elemento representa el número de neuronas en la i-ésima capa oculta. Nosotros hemos considerado redes neuronales de dos capas, ambas con el mismo número de neuronas, de 5, 10, 25 y 50 cada una de ellas. Es decir, el conjunto de búsqueda de esta variable es $\{(5,5),(10,10),(25,25),(50,50)\}$. Este número de capas y de neuronas puede parecer pequeño, pero está justificado en que, al tomar subconjuntos balanceados de ejemplos para entrenar cada estimador, limitamos nuestra muestra a un tamaño muy reducido (1500 elementos, como hemos dicho anteriormente), que no sería suficiente para entrenar una red mucho más compleja. Hemos optamos, por tanto, por entrenar un gran número de redes neuronales mucho más sencillas que contribuyan a la toma de decisiones del bagging.

- $\alpha \in \{0.0001, 0.001, 0.01, 0.1\}$

  Es el índice de regularización l_2 que utilizaremos en el modelo. Los valores sobre los que hemos decidido realizar la búsqueda son $[0.0001, 0.001, 0.01, 0.1]$ para su elección nos hemos basado en este artículo.

El clasificador *MLPClassifier* de la biblioteca *sklearn* solamente soporta la función de pérdida de entropía cruzada, que en el caso de clasificación binaria es:

$$Loss(\hat{y}, y, W) = -y \log \hat{y} - (1-y)\log(1-\hat{y}) + \alpha||W||_2^2$$

Donde $\hat{y}$ es la predicción realizada en el paso hacia adelante, y es la etiqueta de dicho ejemplo, α es el factor de regularización y W es un tensor que contiene todos los pesos de la red.

4.1.4 Balanced Bagging de Support Vector Classifier

Para el uso de modelos basados en máquinas de vectores de soporte en el problema de casificación binaria hemos hecho uso de la clase *SVC* de *sklearn* que es la versión de este modelo empleada para clasificación.

SVM es un algoritmo ampliamente utilizado, sobre todo con kernels no lineales, debido a que gracias a su formulación matemática permite encontrar hiperplanos separadores que son óptimos en cierto sentido apropiado para la clasificación (maximiza la distancia de los vectores soporte al hiperplano). Sumando sus opciones para la regularización, cierto grado de tolerancia al ruido y el uso del truco del kernel (que permite trabajar con datos de una manera equivalente a aplicar transformaciones no lineales sobre ellos) con el kernel de *Radial Basis Functions-Gaussiano*, que hace que el espacio transformado tenga dimensión infinita, lo que da mucha flexibilidad a la hora de elegir la complejidad del modelo.

Al igual que en el resto de casos por un lado hemos dejado fijos los valores de una serie de parámetros como hemos creído conveniente y por otro para decidir el valor de aquellos parámetros hemos realizado búsquedas entre conjuntos de valores para intentar hallar la combinación de estos que mejores resultados ofrezcan.

Dicho esto pasamos a comentar los diferentes parámetros de esta clase y el porqué de haber fijado o evaluado cada uno de ellos. Comenzamos con los parámetros fijados:

- *kernel = 'rbf'*

  Con este parámetro indicamos el tipo de *kernel* que queremos emplear en el algoritmo. En este caso hemos decidido utilizar *'rbf' (Radial Basis Function)* ya que este kernel se suele utilizar cuando no se tiene mucha información a priori del *dataset*. La ecuación del *kernel* es la siguiente:

$$k(x_i, x_j) = e^{-\gamma||x_i - x_j||^2}$$

- *class_ weight = 'balanced'*

  Asignando *'balanced'* a este parámetro indicamos explícitamente que las clases están desbalanceadas y SVC se encarga automáticamente de ajustar los pesos de manera inversamente proporcional a la frecuencia de cada clase entre los datos de entrada.

- *max_ iter=10000*

  Es el máximo número de iteraciones que le permitimos realizar al algoritmo de optimización. Lo fijamos a 10000 por limitaciones computacionales.

Los parámetros que hemos utilizado para realizar la búsqueda son:

- $C \in \{1, 10, 100, 1000, 10000\}$

  Este parámetro representa el valor inverso del índice de regularización l_2 que es la única que este modelo es capaz de aplicar. Los valores que hemos decidido evaluar son, como en el caso anterior, [1,10,100,1000,10000] por los mismos motivos citados previamente.

- $\gamma \in \{0.05, 0.01, 0.005\}$

  Este parámetro sirve para definir la complejidad del modelo. El conjunto de valores entre los que hemos decidido buscar la mejor combinación son [0.05,0.01,0.005] porque se suele fijar como la inversa del número de características, en nuestro caso 100, y hemos decidido buscar valores en torno a dicho valor.

Pasamos ahora a explicar cuál es la función de pérdida que emplea este modelo para encontrar la mejor solución con cada posible combinación de parámetros. Cuando solucionamos un problema de *Machine Learning* utilizando el modelo de SVM lo que se intenta es encontrar unos pesos w y b para los datos de entrada tales que la predicción dada por signo($w^T\phi(x) + b$ sea correcta para la mayoría de datos. Realmente SVC el problema que resuelve es

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i^1}^{n} \zeta_i$$

$$\text{sujeto a } y_i(w^T\phi(x_i) + b) \geq 1 - \zeta_i \text{ con } \zeta_i \geq 0 \quad \forall i = 1, \ldots, n$$

Intuitivamente, intenta maximizar el margen al tiempo que se paga una cierta penalización cuando un dato está mal clasificado, penalización cuya tolerancia controlamos con el parámetro C previamente mencionado.

4.1.5 Balanced Random Forest

En este caso no nos ha hecho falta añadir *BalancedBaggingClasiffier* ya que la biblioteca *imblearn* incorpora un clasificador usando *Random Forests* que internamente lidia con el problema del balanceo. Este clasificador es *BalancedRandomForestClassifier*. Antes de mostrar los parámetros con los que hemos trabajado cabe resaltar que en *Random Forests* todos aquellos parámetros que están relacionados con podar/limitar el tamaño de los árboles no los hemos cambiado (no podamos ni limitamos el tamaño del árbol), puesto que el interés de podar los árboles es reducir su varianza al tener en cuenta únicamente las variables que separan la muestra más claramente. En meta-modelos tipo ensemble lo que necesitamos es mejorar todo lo posible los resultados individuales de cada modelo base, puesto que al combinar sus resultados estamos reduciendo implícitamente su varianza.

A este experimento le añadiremos un elemento extra. Podríamos haberlo realizado con cualquiera de los modelos anteriores, pero por simplicidad lo restringiremos únicamente a los Random Forest. Dicho cambio consiste en calcular la **probabilidad** devuelta por cada modelo **de que un elemento pertenezca a una clase** (en este caso, usaremos la clase *neg* como referencia), y estudiar como varía el comportamiento general del modelo cuando cambiamos el umbral de admisión a dicha clase, esto es, la probabilidad por encima de la cual aceptamos que dicho ejemplo pertenece a la clase *neg*, y no a la clase *pos*. Claramente, el umbral estandar es 0.5.

La motivación detrás de este experimento consiste en incluir un mecanismo de control de los errores asociados a etiquetar los elementos de tipo *neg* como de tipo *pos*, ocasionado por el siguiente fenómeno: al balancear las muestras de entrenamiento, esperamos encontrar un error similar en ambas clases. Esto no es necesariamente bueno, puesto que podemos encontrarnos con unas **precisiones** ($\frac{TP}{TP+FP}$) **muy bajas en la clase minoritaria**, esto es, si la clase mayoritaria tiene el mismo porcentaje de error que la clase minoritaria, podemos encontrarnos que el número de elementos de la mayoritaria etiquetados como de la minoritaria exceda incluso al número de elementos que verdaderamente pertenecen a la clase minoritaria. Esto no es deseable, y por ello añadiremos el mecanismo anteriormente explicado en un intento de controlar este descenso de la precisión. Un método similar se utiliza en este experimento en Kaggle para el mismo dataset, aunque de forma distinta, para combatir el desbalanceo de las clases.

Dada la explicación anterior, entendemos que tiene sentido experimentar con valores por debajo de 0.5, aunque más o menos cercanos a este número. Los valores para este **umbral** con los que hemos experimentado son **0.5, 0.45, 0.4, 0.35, 0.3**.

A continuación mostramos los parámetros que hemos dejado fijos en el proceso de entrenamiento:

- *bootstrap = True*

  El hecho de activar el bootstraping habilita el remuestreo del dataset. Por tanto, si el parámetro es False se utiliza el dataset completo para construir cada uno de los árboles y en nuestro caso nos hemos decantado por tomar muestras mas pequeñas.

- *criterion = 'entropy'*

  Este parámetro indica la técnica que se quiere utilizar para indicar a impureza de un nodo hoja del árbol. Como estamos trabajando con un problema de clasificación binaria sabemos que las diferentes medidas de impureza (y por tanto los diferentes valores del parámetro) son equivalentes y por ello elegimos *'entropy'*.

- *max_ depth = None.*

  Es la profundidad máxima del árbol. Lo hemos fijado a None para que se expanda hasta que las hojas no se puedan particionar mas.

- *max_features = 'sqrt'*

  Con este parámetro podemos controlar el número de caracteristicaas a cosiderar en la búsqueda de la mejor partición del nodo. En nuestro caso hemos decidido que sea la raiz cuadrada del número total de características de los datos, puesto que se ha observado empíricamente que obtiene buenos resultados para clasificación, constituyendo un buen valor por defecto.

- *min_ samples_ leaf = 1*

  Con este parámetro se puede indicar el número mínimo de elementos que tiene que haber en un nodo hoja. En nuestro caso hemos decidido que con un elemento por hoja es suficiente, puesto que no queremos limitar el crecimiento del árbol.

- *min_ samples_ split=2*

  Este es el parámetro que controla el número de elementos que tiene que haber en un nodo para decidir particionarlo. Lo dejamos a 2 para no limitar la estrategia de separación de nodos del árbol.

Por otro lado, los parámetros cuyos valores hemos ido buscando, aparte del umbral explicado más arriba, son:

- *n_ estimators* $\in \{100, 200, 4000, 750, 1000\}$

  Este parámetro indica el número de árboles que queremos en el bosque. Los valores entre los que vamos a buscar el óptimo son $[100, 200, 4000, 750, 1000]$.

- *ccp_ alpha* $\in \{0, 0.1, 0.001, 0.0001\}$

  Este parámetro representa la penalización que se le da a la complejidad del árbol, se puede ver como el ratio de regularización de estos modelos. En nuestro caso hemos decidido buscar el mejor valor en el conjunto $[0, 0.1, 0.001, 0.0001]$

- *umbral* $\in \{0.5, 0.45, 0.4, 0.35, 0.3\}$

  Probabilidad estimada de pertenecer a la clase *neg* por encima de la cual etiquetamos a un ejemplo con dicha clase y no con la clase *pos*.

Para reducir el coste computacional del experimento, hemos dividido el *grid search* utilizado para probar con las distintas combinaciones de hiperparámetros (un total de $5 \cdot 4 \cdot 5 = 100$ combinaciones distintas) en dos *grid searchs* diferentes: uno en el que buscamos un valor apropiado para regularización usando un umbral estandar de 0.5 (20 ejecuciones de *5-fold cross-validation*) y otro en el que restringimos el valor de regularización y de número de estimadores a aquellos que hayan dado mejores resultados (otras 12 ejecuciones).

La función de pérdida que utiliza Random Forest es la siguiente:

$$R(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|$$

Donde T es el árbol que estamos evaluando, $|T|$ es el número de nodos terminales del árbol, $|N_m|$ es el número de datos que han caído en el nodo terminal m, α es el coeficiente que penaliza la complejidad del árbol (se utiliza para regularizar) y la función $Q_m(T)$ la medida de impureza del nodo terminal m.

En nuestro caso la forma de medir la impureza es un poco indiferente ya que al estar tratando con un problema de clasificación binaria los tres métodos de los que disponemos (*missclasification error, gini y entropy*) son completamente equivalentes. Por tanto en nuestro caso elegimos el criterio de *entropy*:

$$Q_m(T) = -\sum_{k}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$$

Donde $\hat{p}_{mk}$ es la proporción de elementos etiquetados con el identificador de la clase k en el nodo m.

4.2 Resultados

Este apartado de la memoria lo dedicaremos a comentar los resultados que hemos obtenido tras entrenar cada no de los diferentes modelos para cada una de las combinaciones de parámetros que hemos evaluado. Mostraremos tanto el *accuracy* obtenido mediante el uso de *cross-validation* y el SCORE resultante tras aplicar la métrica que hemos definido. Además, resaltaremos la mejor de los combinaciones obtenidas mostrando también su matriz de confusión.

4.2.1 Balanced Bagging de Regresion Logística

En la tabla 2 aparecen reflejados los resultados que hemos obtenido en la búsqueda que hemos realizado sobre el bagging de regresión logística, marcando en negrita el mejor de ellos. El primer hecho que llama la atención es la mejoría que supone el utilizar la combinación *lbfgs*+l_2 frente a *saga*+l_1, puesto que todos los SCORE de uno son menores que todos los del otro y por tanto mejores según nuestro criterio. No solo eso, las *accuracy* obtenidas también son mejores. En cualquier caso, ambos métodos obtienen mejores puntuaciones SCORE que la RL sin balancear (tabla 1), puesto que buscan aprender a clasificar correctamente tanto la clase mayoritaria como la minoritaria.

En la matriz de confusión de la figura 10 podemos ver el ratio de datos bien etiquetados frente al de mal etiquetados en función de cada una de las etiquetas. Esto nos hace apreciar visualmente el hecho de haber elegido la métrica SCORE para la elección del mejor de los resultados.

También cabe destacar como cambian la precisión y el *recall* al usar baggings con muestreo balanceado. El *recall* de la clase minoritaria crece considerablemente, puesto que nuestro modelo pasa a intentar clasificar correctamente la clase minoritaria. La precisión, sin embargo, sufre los efectos explicados en la sección 4.1.5.

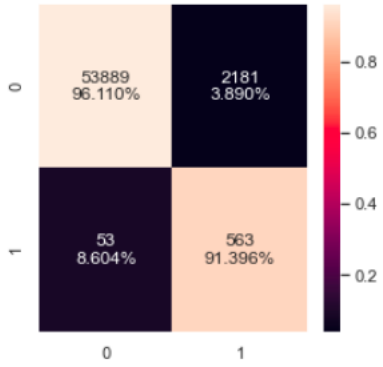
Bagging de RL				
Solver	**Tipo de Regularización**	**C**	**ACCURACY**	**SCORE**
saga	L1	1	0.8775	77750
saga	L1	10	0.8651	84800
saga	L1	100	0.8634	85740
saga	L1	1000	0.8632	85860
saga	L1	10000	0.8632	85870
lbfgs	L2	1	0.9672	50910
lbfgs	L2	10	0.9677	52100
lbfgs	L2	100	0.9679	53000
lbfgs	L2	1000	0.9679	53490
lbfgs	**L2**	**10000**	**0.9606**	**48310**

Table 2: Resultados para el bagging balanceado constituido por modelos tipo regresión logística.

En la figura 11 observamos como se distribuyen las predicciones hechas por los mejores modelos de tipo RL y Bagging de RL. RL tiene una proporción de puntos naranjas (clase *pos*) mucho más parecida a la original, aunque ya sabemos que no clasifica esta clase correctamente. La baja precisión comentada anteriormente se traduce en gran cantidad de puntos naranjas que pueden verse en las predicciones del Bagging de RL.

4.2.2 Bagging de Neural Networks

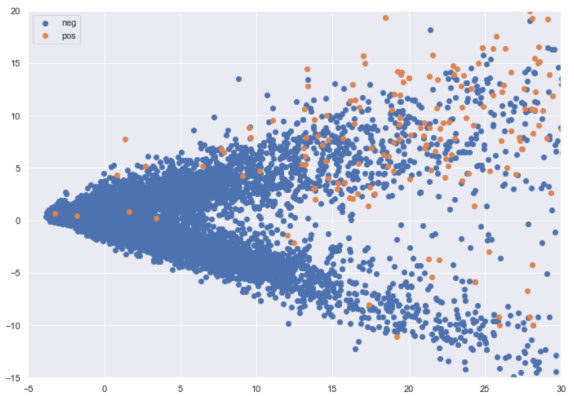
En la tabla 3 aparecen los resultados que hemos obtenido en la búsqueda sobre el Balanced Bagging de Redes Neuronales, marcando en negrita el mejor de ellos. En esta tabla se aprecia que estabamos en lo cierto cuando decidimos utilizar capas con pocas neuronas en cada una de ellas ya que no solo los resultados en general rondan las cifras que se alcanzan con otros sino que dentro de los resultados del propio modelo vemos que las capas más pequeñas ofrecen resultados mejores a los de las más grandes.



Precision *neg* : 0.99901
Recall *neg* : 0.9611

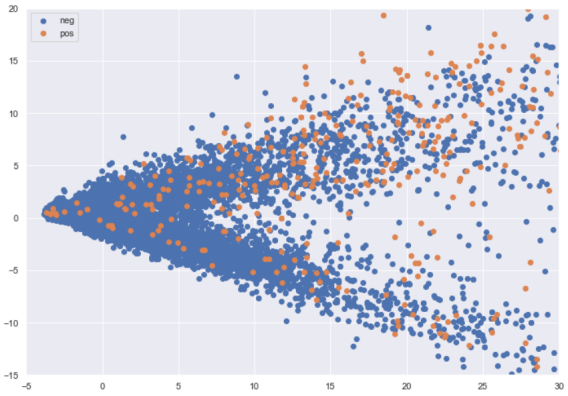
Precision *pos* : 0.20517
Recall *pos* : 0.91396

Figure 10: Matriz de confusión del resultado ofrecido por Balanced Bagging de Regresión Logística. Medidas de *precision* y *recall*.



Regresión Logística

Bagging de Regresión Logística



Etiquetas originales

Figure 11: Proyección del conjunto de entrenamiento sobre las dos componentes principales. Comparación entre las predicciones de RL (izquierda), Bagging de RL (centro) y las etiquetas originales (derecha).

La matriz de confusión correspondiente (figura 12) muestra un mejor *recall* que en el bagging de RL, pero una precisión menor. Esto quiere decir que mejoramos a la hora de clasificar los elementos de la clase minoritaria, pero que nuestra confianza en que un ejemplo que clasificamos como *pos* pertenezca realmente a esa clase es cada vez menor: hay un número muy elevado de falsos positivos, concretamente 3113, frente a los 616 ejemplos que realmente pertenecen a esta clase.

Bagging de Redes Neuronales			
Layers	**α**	**ACCURACY**	**SCORE**
(5,5)	0.0001	0.9361	48990
(5,5)	0.001	0.9433	47840
(5,5)	0.01	0.966	49190
(10,10)	0.0001	0.9422	48460
(10,10)	**0.001**	**0.9445**	**47630**
(10,10)	0.01	0.9661	49600
(25,25)	0.0001	0.9463	48590
(25,25)	0.001	0.9468	48310
(25,25)	0.01	0.9653	48600
(50,50)	0.0001	0.9491	52410
(50,50)	0.001	0.9491	52360
(50,50)	0.01	0.9651	49190

Table 3: Resultados para el Bagging de Redes Neuronales sencillas con submuestreo balanceado.

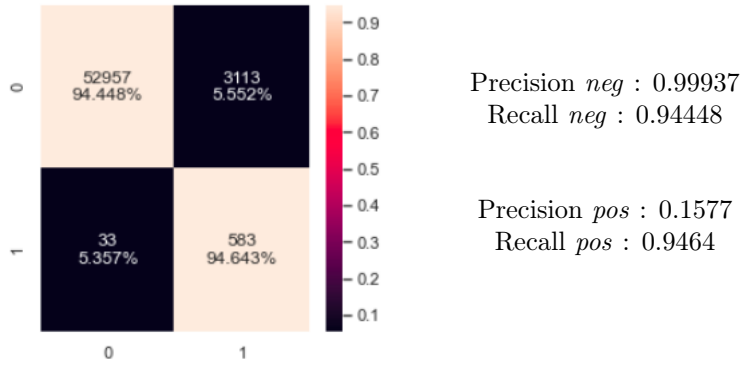


Figure 12: Matriz de confusión del resultado ofrecido por Balanced Bagging de Neural Network.

4.2.3 Balanced Bagging de SVC

En la tabla 4 aparecen los resultados que hemos obtenido en la búsqueda sobre el balanced bagging de máquinas de vectores soporte que hemos llevado a cabo, marcando en negrita el mejor de ellos.

En la matriz de confusión de la figura 13 vemos que aunque clasifica la clase minoritaria algo peor que el modelo de redes neuronales (8 falsos negativos más) ha reducido en 300 el número de falsos positivos. Aún así seguimos considerando que la solución de la red neuronal es mejor que la de SVM debido a la gran penalización que nuestra métrica de evaluación asigna a un dato mal clasificado de la clase minoritaria.

Los mejores resultados obtenidos mediante SVC bastante cercanos a los del Bagging de RL usando *lbfgs*+l_2, perdiendo algo de SCORE pero mejorando notablemente el *recall*. Queremos señalar, en detrimento del SVC-RBG, lo sensible que es este a modificar el parámetro γ. Cambiar este parámetro de 0.1 a 0.5 nos hace perder en torno a un 13% de exactitud total y duplicar el SCORE que buscamos minimizar. No hemos visto una variabilidad tan alta en los Baggings de MLPs ni en los de Regresión Logística ya mencionados.

4.2.4 Balanced Random Forest

En esta sección mostramos los resultados del primer *grid search* de ajuste de hiperparámetros para Random Forest, que usaremos para seleccionar aquellos valores de regularización y número de estimadores que se tendrán en cuenta en el siguiente apartado. Estos resultados se ven reflejados en la tabla 5, donde hemos remarcado en negrita la mejor de todas las combinaciones según el coste mostrado en SCORE.

Bagging de SVC con kernel RBF			
C	**γ**	**ACCURACY**	**SCORE**
1	0.05	0.8197	110540
1	0.01	0.9334	53950
1	0.005	0.9495	50170
10	0.05	0.8348	101970
10	0.01	0.9341	53520
10	0.005	0.9489	49520
100	0.05	0.8351	101800
100	0.01	0.9679	53210
100	0.005	0.9494	48780
1000	0.05	0.8351	101800
1000	0.01	0.9347	53210
1000	**0.005**	**0.9495**	**48710**
10000	0.05	0.8351	101800
10000	0.01	0.9347	53210
10000	**0.005**	**0.9495**	**48710**

Table 4: Resultados para el bagging balanceado constituido por modelos de tipo SVC con kernels RBF.

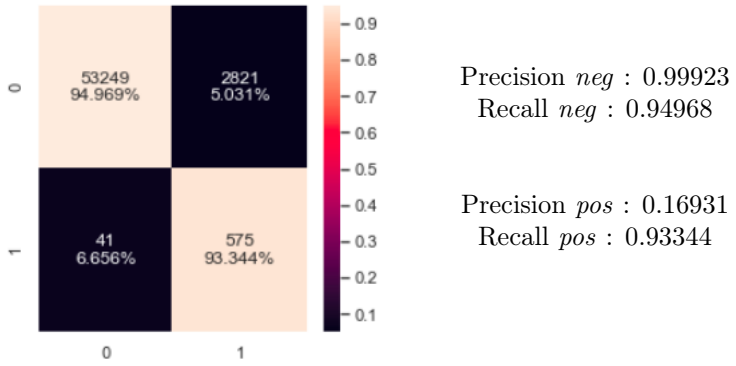


Figure 13: Matriz de confusión del resultado ofrecido por el Balanced Bagging de SVC. Medidas de *precision* y *recall*.

En la tabla vemos que las soluciones obtenidas no son muy sensibles a variaciones entre valores muy pequeños del parámetro α ya que el coste obtenido mediante SCORE no varía con dicho parámetro entre los tres valores mas pequeños de los 4 que estudiamos. Es por este motivo que decidimos fijar el parámetro α a 0 en el siguiente paso del experimento. También hemos limitado el número de árboles a aquellos valores que se encuentran en torno a 400, con el que hemos obtenido el mejor SCORE, puesto que no parece alterar los resultados en gran medida.

Hasta el momento, Random Forest obtiene los mejores resultados, aunque no por mucha diferencia sobre las redes neuronales. En caso de empeorar nuestros resultados al cambiar el umbral, tomaremos el usado por defecto (0.5) con 400 estimadores y $\alpha = 0$.

4.2.5 Balanced Random Forest (Prob)

En este último caso mostramos (reflejados en la tabla 15) los resultados obtenidos usando el modelo de Random Forest asignando diversos umbrales a la probabilidad mínima aceptada para clasificar un dato en la clase *neg*. Como hemos comentado previamente, fijaremos el parámetro $\alpha = 0$ e incluimos la probabilidad umbral en el *grid search*.

Observamos que reducir el umbral de admisión en la clase mayoritaria ayuda a reducir el SCORE total. Si comparamos la matriz de confusión de la figura 15 con la del Random Forest anterior (figura 14)

Random Forest			
Nº de estimadores	**α**	**ACCURACY**	**SCORE**
100	0.01	0.9388	48880
100	0.001	0.9416	47340
100	0.0001	0.9416	47340
100	0.0	0.9416	47340
200	0.01	0.9383	49210
200	0.001	0.9408	47780
200	0.0001	0.9408	47780
200	0.0	0.9408	47780
400	0.01	0.939	48290
400	**0.001**	**0.941**	**47160**
400	**0.0001**	**0.941**	**47160**
400	**0.0**	**0.941**	**47160**
750	0.01	0.9384	48660
750	0.001	0.9407	47360
750	0.0001	0.9407	47360
750	0.0	0.9407	47360
1000	0.01	0.9383	48680
1000	0.001	0.9407	47330
1000	0.0001	0.9407	47330
1000	0.0	0.9407	47330

Table 5: Resultados para el Random Forest con submuestreo balanceado.

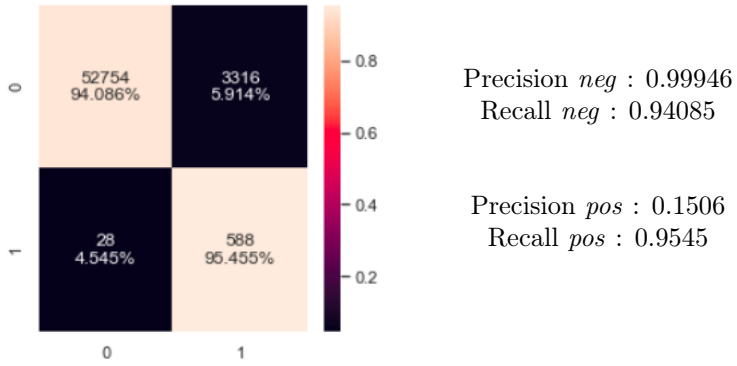


Figure 14: Matriz de confusión del resultado ofrecido por Balanced Bagging de Random Forest. Medidas de *precision* y *recall*.

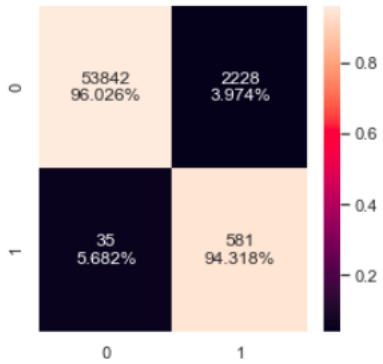
podemos ver que este método tiende a equilibrar mejor la precisión de la clase minoritaria conservando un *recall* casi tan alto como antes de modificar el umbral. Este modelo es el que obtiene la mejor solución de todas con un coste de 39780.

En la figura 16 apreciamos visualmente los efectos de esta técnica. Vemos como bajar el filtro de admisión de un elemento a la clase *neg*, asignando esta clase a aquellos puntos con una probabilidad estimada del 35% o más de pertenecer a la misma la cantidad de puntos naranjas se reduce notablemente, lo que se traduce en menos falsos positivos.

Concluimos seleccionando el Random Forest balanceado con los parámetros vistos anteriormente como nuestro modelo final. En el apartado siguiente evaluaremos sus resultados con el conjunto de test.

Random Forest filtrando probabilidades			
Nº de estimadores	**Umbral**	**ACCURACY**	**SCORE**
200	0.3	0.9651	39860
200	0.35	0.9595	41070
200	0.4	0.9533	42670
200	0.45	0.9467	44420
400	0.3	0.9656	41070
400	0.35	0.96	41270
400	0.4	0.9536	42470
400	0.45	0.9475	44440
750	0.3	0.9657	40490
750	**0.35**	**0.9601**	**39780**
750	0.4	0.9533	43120
750	0.45	0.9471	44680

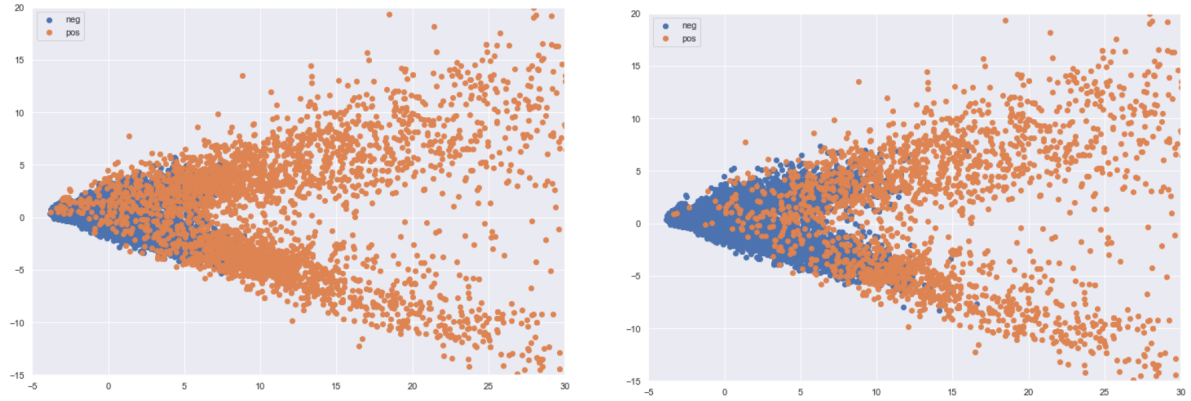
Table 6: Resultados para el Random Forest con submuestreo balanceado, filtrando los ejemplos etiquetados como *pos* con una confianza por debajo de cierto umbral.



Precision *neg* : 0.99935
Recall *neg* : 0.96026

Precision *pos* : 0.20683
Recall *pos* : 0.94318

Figure 15: Matriz de confusión del resultado ofrecido por BalancedRandom Forest (Prob). Medidas de *precision* y *recall*.



Random Forest

Random Forest con clasificación ponderada

Figure 16: Comparación entre las predicciones de RF (izquierda) y de RF filtrando predicciones tomando un umbral de 0.35 (derecha).

5 Test

En esta sección evaluaremos el modelo final seleccionado, mediante el conjunto de test proporcionado junto al resto del dataset.

A modo de comparación, evaluaremos también otros de los modelos obtenidos en los *grid search* de la sección anterior. Es importante destacar que en una situación práctica probablemente no dispondríamos de este conjunto de test, y por tanto no habría forma de realizar una comparación como la siguiente. El **modelo final es única y exclusivamente** aquel que ha logrado un menor SCORE en validación cruzada. Evaluar el resto de modelos con el conjunto de test tiene como fin ayudarnos a argumentar que se ha obtenido la mejor de las posibles soluciones para la muestra dada. Los resultados en test de los mejores modelos (en validación cruzada) tipo Bagging de LR, Bagging de RRNN, Bagging de SVC y Random Forest, reentrenados a partir del conjunto de entrenamiento completo, pueden verse ordenados en la figura 17.

En este sentido, concluimos que el modelo que mejor puntuación ha logrado en validación vuelve a ser mejor al comparar los resultados en fase de test. Random Forest logra el SCORE más bajo (recordamos que esta métrica es una penalización ponderada de los falsos positivos y los falsos negativos, poniéndo énfasis en los segundos) además de una precisión más alta que el resto gracias al método de filtrado añadido al modelo base de Random Forest, aunque aparenta ser bastante baja de todos modos.

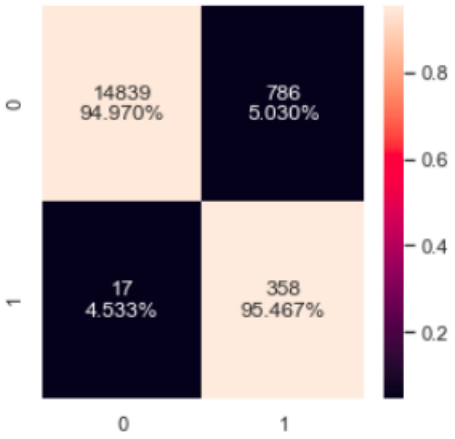
Utilizar un conjunto de test de este tamaño para evaluar nuestra hipótesis final nos permite asumir que el error fuera de la muestra debe ser similar al error medido sobre este conjunto de datos, asumiblemente independiente del conjunto de entrenamiento pero que sigue su misma distribución de probabilidad. Por tanto, podemos generalizar las medidas de SCORE, *precision* y *recall* de test como medidas del error fuera de la muestra. Recalcamos que, en este problema, la exactitud es una medida engañosa. Por tanto, consideramos apropiado entender E_{out} como el **SCORE esperado fuera de la muestra**, por encima del error de clasificación.

Por otro lado, podemos asegurar que el resultado que hemos dado como solución al problema de *Machine Learning* que se nos ha planteado aparenta ser relativamente bueno. Esto lo podemos sospechar porque en nuestro caso tenemos la posibilidad de comparar nuestra solución con las de otras personas que se han enfrentado al mismo problema utilizando la misma métrica de evaluación. Estos resultados se pueden ver reflejados en el ranking que hay disponible en la documentación sobre el dataset en el respositorio oficial.

TOP 3	SCORE	Falsos Positivos	Falsos Negativos
Camila F. Costa and Mario A. Nascimento	9920	542	9
Christopher Gondek, Daniel Hafner and Oliver R. Sampson	10900	490	12
Sumeet Garnaik, Sushovan Das, Rama Syamala Sreepada and Bidyut Kr. Patra	11480	398	15

Podemos ver claramente que, aunque no hayamos alcanzado los valores de SCORE que aparecen reflejados en dicho ranking, nos hemos quedado muy cerca del tercer puesto que, asumimos, es una solución considerabemente buena. Además, si buscamos un modelo que minimice los falsos negativos manteniendo una cantidad aceptable de falsos positivos, nuestro modelo podría competir con los puestos segundo y tercero, en tanto que obtiene menos falsos negativos.

Bagging de Regresión Logística

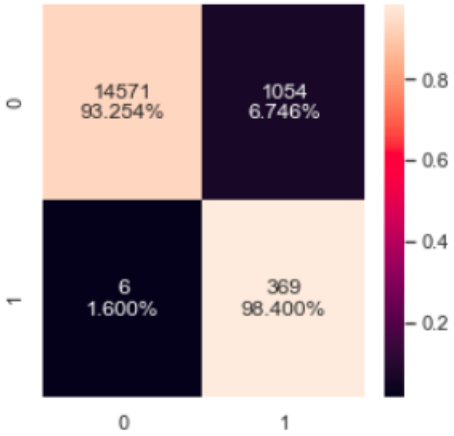


Accuracy: 0.9498
SCORE: 16360

Precision *neg* : 0.99885
Recall *neg* : 0.94969

Precision *pos* : 0.3129
Recall *pos* : 0.9546

Bagging de Redes Neuronales

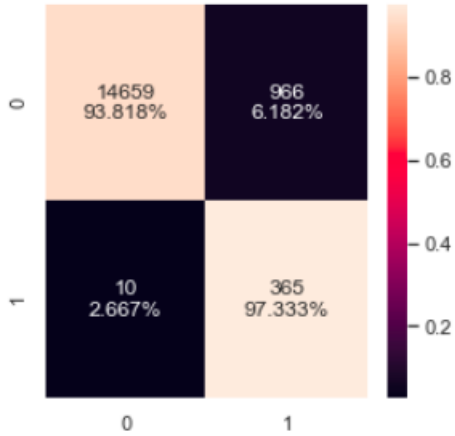


Acccuracy: 0.9338
SCORE: 13540

Precision *neg* : 0.99958
Recall *neg* : 0.93254

Precision *pos* : 0.25931
Recall *pos* : 0.984

Bagging de SVC

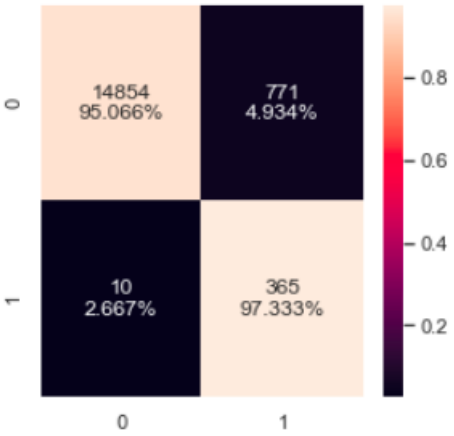


Accuracy: 0.939
SCORE: 14660

Precision *neg* : 0.9993
Recall *neg* : 0.93817

Precision *pos* : 0.2742
Recall *pos* : 0.9733

Random Forest



Accuracy: 0.9512
SCORE: 12710

Precision *neg* : 0.9993
Recall *neg* : 0.9506

Precision *pos* : 0.3213
Recall *pos* : 0.9733

Figure 17: Resultados en test.