Inteligencia de Negocio (2019-2020)
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Análisis Predictivo Mediante Clasificación

Grupo de prácticas: martes
Email: simondelosbros@correo.ugr.es

Simón López Vico

20 de septiembre de 2020

Índice

1. Introducción

En esta práctica nos centraremos en el estudio del comportamiento de diferentes algoritmos de clasificación mediante el diseño experimental apropiado y el análisis comparado de resultados, así como en la extracción conclusiones a partir del conocimiento aprendido mediante estos algoritmos para comprender las relaciones entre los atributos que favorecen una determinada clase.

Trabajaremos sobre un dataset de Taarifa y el Ministerio de Agua de Tanzania, el cuál contará de 59400 instancias, cada una de ellas con 39 atributos referentes a qué tipo de bomba se está usando, cuándo se instaló, cómo se administra, su ubicación, datos sobre la cuenca geográfica, tipo de extracción, coste del agua, etc. Nuestro conjunto de datos constará de tres clases que determinan el estado de una bomba de extracción de agua: funcional, con necesidad de alguna reparación y no funcional. El porcentaje de instancias etiquetadas con cada clase no es el mismo para las tres: un 55 % de instancias son funcionales, un 7 % necesitan reparación y un 38 % son no funcionales (aproximadamente).

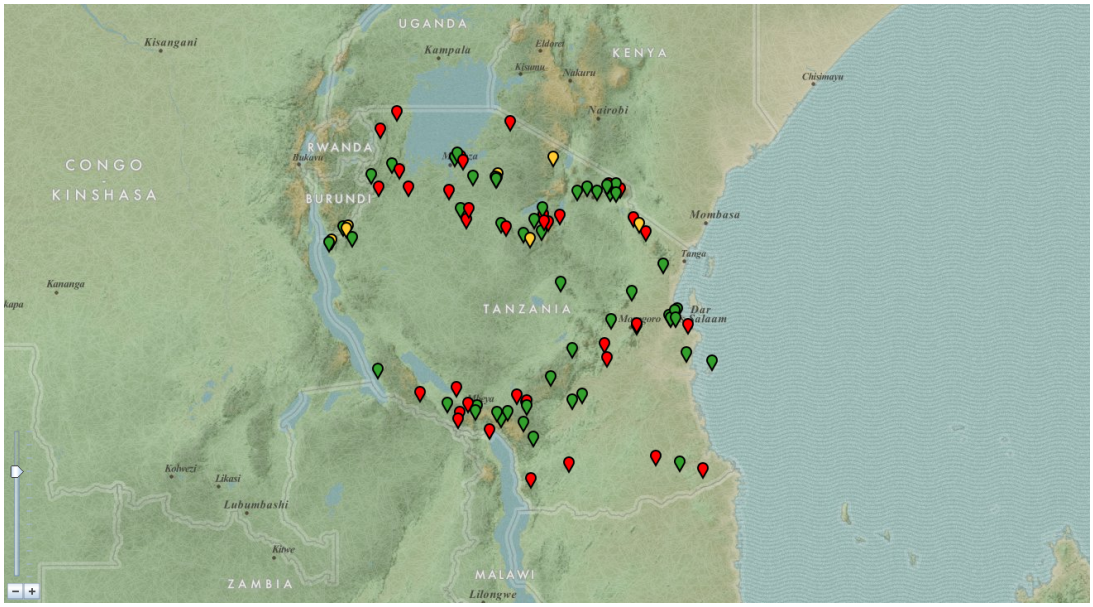


Figura 1.1: Ubicación de 100 bombas de agua escogidas aleatoriamente del dataset. Verde = funcional ; amarillo = reparación; rojo = no funcional.

Además, algunas de las instancias contienen valores perdidos, no asignados, los cuales tendremos que inferir para la ejecución de ciertos algoritmos o como preprocesado de los datos para lograr un mejor entrenamiento. Los atributos con valores perdidos (en orden descendente) son: scheme_name (28174), scheme_management (3877), installer (3877), funder (3662), public_meeting (3334), permit (3056), subvillage (371) y wpt_name (20).

Para todos los algoritmos, se ha usado *Cross Validation* con 5 particiones, marcando la opción *Stratified Sampling* para que el número de instancias etiquetadas con una determinada clase sea similar en todas las particiones, y estableciendo como semilla el valor 26504148.

Los algoritmos usados para el desarrollo de la práctica han sido: **C4.5** (árboles de decisión), **Random Forest**, **REPtree Bagging** (bagging), **Gradient Boosted** (boosting), **K-NN** (basado en instancias), **Naive Bayes** (probabilístico) y **Rprop MLP** (redes neuronales).

2. Resultados obtenidos

Comenzaremos leyendo nuestro dataset utilizando el nodo File Reader, el cual será enlazado con todos los metanodos creados para los distintos algoritmos. Como hemos comentado anteriormente, ejecutaremos todos nuestros algoritmos con validación cruzada de 5 particiones, utilizando los nodos *X-Partitioner* y *X-Aggregator*, que configuraremos de la siguiente manera:

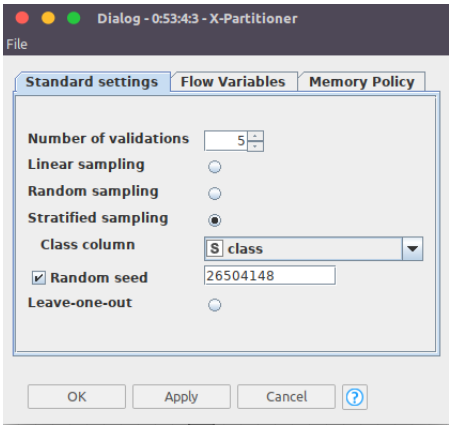


Figura 2.1: Configuración del nodo *X-Partitioner* para todos los algoritmos.

Por otra parte, será necesario un ligero preprocesado para la ejecución de algunos algoritmos; estas modificaciones sobre nuestro dataset serán las mismas para todos los algoritmos que las necesiten.

- *Category to Number:* convierte todos los valores nominales en numéricos. Configuraremos este nodo haciendo que transforme todos los atributos de cada instancia salvo la clase y estableciendo como número máximo de valores 37500. Aplicado en: K-NN, Rprop MLP y REPtree Bagging.
- *Missing Value:* infiere con técnicas estadísticas los valores perdidos en el dataset. Los valores numéricos perdidos los solucionaremos con la media y los valores nominales con el valor más frecuente. Aplicado en: Rprop MLP.

- *Normalizer:* (solo con valores numéricos) transforma los datos para llevarlos al intervalo [0,1]. Será necesario para un correcto funcionamiento de K-NN y Rprop MLP. Aplicado en: K-NN y Rprop MLP.

Una vez terminada la ejecución de cada uno de los algoritmos, *X-Aggregator* devolverá una tabla que contendrá el dataset completo junto a la predicción de cada instancia. Estos datos serán procesados por el metanodo *Precision Measures*, que contendrá para cada algoritmo un *Scorer* que nos proporcionará la matriz de confusión y los datos estadísticos de nuestro modelo. Dichos datos serán necesarios para calcular la precisión de las predicciones realizadas.

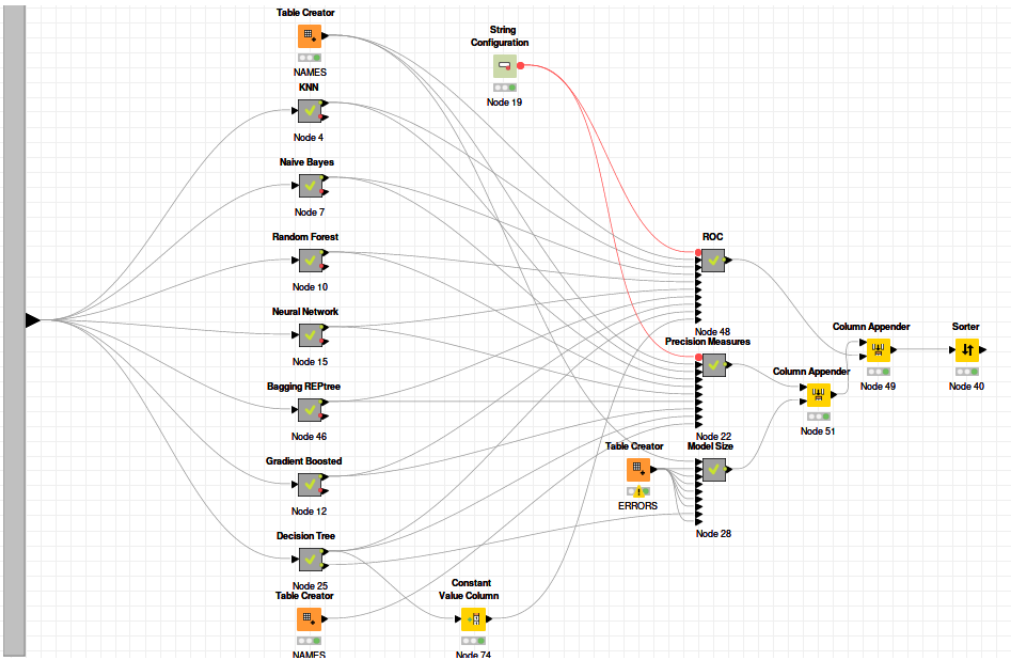


Figura 2.2: Flujo de trabajo para los diferentes algoritmos ejecutados sin preprocesado.

Para calcular el *accuracy* y el *G-mean* del modelo obtenido realizaremos los siguientes cálculos usando el nodo *Math Formula*:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{2.1}$$

$$G - mean = \sqrt{TPR * TNR} \tag{2.2}$$

Por otra parte, utilizando la distribución normalizada de la clase positiva de nuestro conjunto de datos (*non functional*) calcularemos la curva ROC y el área bajo ella. Todo esto será procesado en el metanodo *ROC*.

2.1. K-NN

Comenzaremos comentando los principales nodos utilizados para el correcto funcionamiento del algoritmo. En adelante, si vuelven a aparecer dichos nodos, se dará por entendido su funcionamiento.

El flujo de trabajo para la ejecución de este algoritmo será el siguiente:

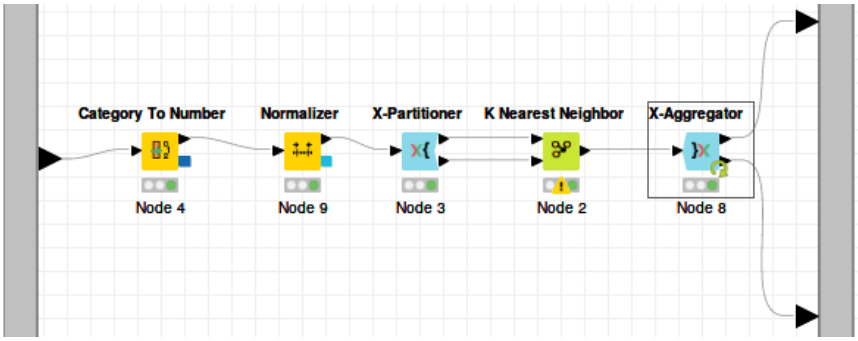


Figura 2.3: Workflow del algoritmo K-NN.

Para empezar, ya que K-NN es un algoritmo basado en la distancia entre las diferentes instancias del conjunto de datos, debemos de pasar los valores categóricos a numéricos; para ello, utilizaremos el nodo *Category to Number* comentado anteriormente. Tras ello hemos de normalizar nuestros datos, pues el algoritmo de los vecinos más cercanos no tiene sentido sin normalización. Finalmente haremos las 5 particiones y aplicaremos el algoritmo con $K = 3$ y sin dar peso a los vecinos por su distancia. El *warning* que nos aparece es debido a los valores perdidos y por tanto el algoritmo se ejecutará obviando las instancias que no tienen todos sus atributos definidos.

Las predicciones obtenidas por K-NN se procesarán en los metanodos *ROC* y *Precision Measures* donde se calcularán los valores para comparar los algoritmos más adelante. Uniendo todos los resultados obtenidos tendremos la siguiente tabla:

	TP	**FP**	**TN**	**FN**	**TPR**	**TNR**	**PPV**	**Accuracy**	**F1-score**	**G-mean**	**Recall**	**AUC**
KNN	7085	2003	16039	2631	0,729	0,889	0,78	0,833	0,754	0,805	0,729	0,543

2.2. Naive Bayes

A continuación vemos el flujo de trabajo del algoritmo Naive Bayes.

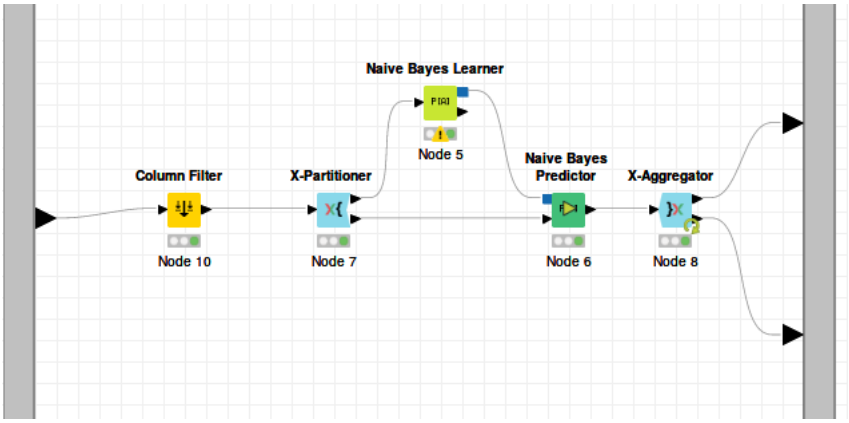


Figura 2.4: Workflow del algoritmo Naive Bayes.

Para la ejecución correcta de este algoritmo comenzamos aplicando un *Column Filter* para eliminar los atributos nominales que tienen una gran cantidad de valores distintos ya que el nodo *Naive Bayes Learner* tendrá fijo un número máximo de valores nominales únicos por atributo (20). Tras ello se realizan las 5 particiones y se ejecuta Naive Bayes con una probabilidad por defecto de 0.0001, una desviación típica mínima de 0.0001 y un umbral de desviación típica de 0.0.

La tabla con los resultados de Naive Bayes es la siguiente:

	TP	FP	TN	FN	TPR	TNR	PPV	Accuracy	F1-score	G-mean	Recall	AUC
Naive Bayes	14739	7170	29406	8085	0,646	0,804	0,673	0,743	0,659	0,721	0,646	0,794

2.3. Random Forest

El metanodo de Random Forest contiene lo siguiente:

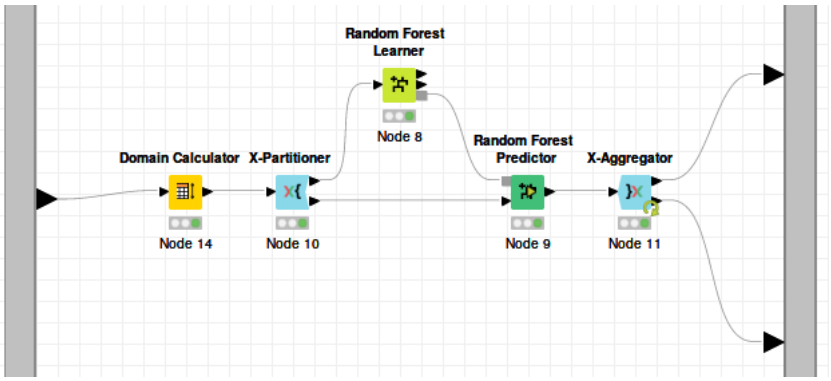


Figura 2.5: Workflow del algoritmo Random Forest.

Comenzamos modificando nuestro dataset con el nodo *Domain Calculator*, que escanea los datos y actualiza la lista de valores posibles y/o los valores mínimos y máximos de las columnas seleccionadas. Esto es necesario para conseguir un correcto funcionamiento de Random Forest.

El algoritmo se ejecuta con un valor máximo de 10 niveles de profundidad y utilizando 100 modelos de árbol. La tabla con los resultados de Random Forest es la siguiente:

	TP	FP	TN	FN	TPR	TNR	PPV	Accuracy	F1-score	G-mean	Recall	AUC
Random Forest	11701	1912	34664	11123	0,513	0,948	0,86	0,781	0,642	0,697	0,513	0,855

2.4. RProp MLP (redes neuronales)

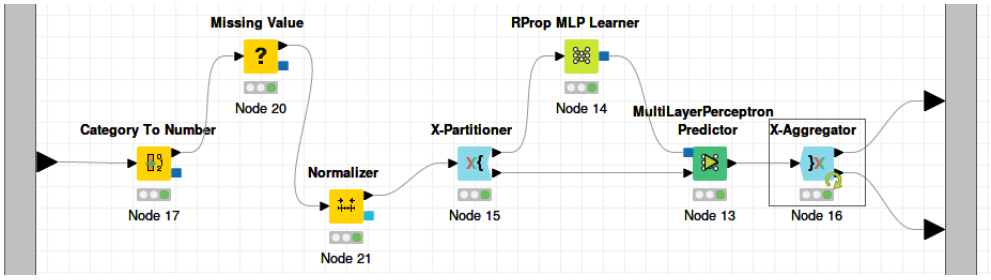


Figura 2.6: Workflow del algoritmo RProp MLP.

Las redes neuronales necesitan trabajar con valores numéricos normalizados y con todos sus atributos definidos, es decir, sin valores perdidos. Por tanto aplicamos el preprocesado correspondiente a ello antes de generar las particiones para el aprendizaje. Usaremos un número máximo de 100 iteraciones, 1 capa escondida y 10 neuronas por capa. La tabla con los valores obtenidos es:

	TP	FP	TN	FN	TPR	TNR	PPV	Accuracy	F1-score	G-mean	Recall	AUC
Neural Network	13736	6135	30441	9088	0,602	0,832	0,691	0,744	0,643	0,708	0,602	0,793

2.5. REPtree Bagging

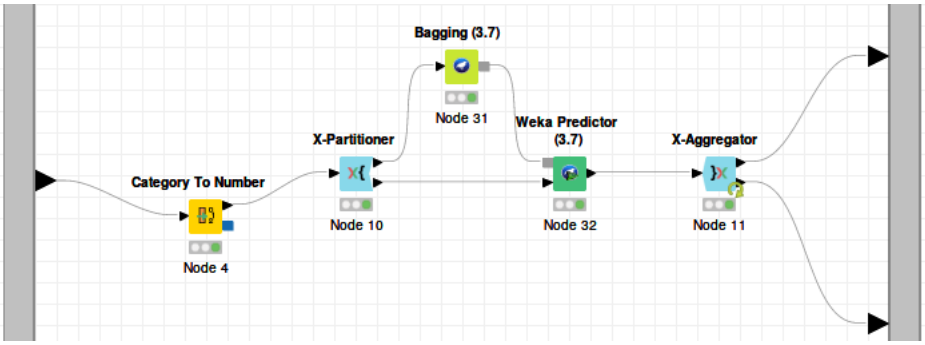


Figura 2.7: Workflow del algoritmo REPtree Bagging.

Ya que el nodo usado para ejecutar Bagging puede hacer clasificación o regresión dependiendo del modelo de aprendizaje que escojamos, se nos exige trabajar con valores numéricos, volviendo a utilizar para ello *Category to Number*. El clasificador escogido para la ejecución de Bagging ha sido REPtree, un clasificador de tipo árbol que toma rápidas decisiones. Construye un árbol de decisión usando ganancia/variación de información y lo poda usando *pruning*.

Con la ejecución del algoritmo se devuelve la siguiente tabla:

	TP	FP	TN	FN	TPR	TNR	PPV	Accuracy	F1-score	G-mean	Recall	AUC
Bagging	17077	3236	33340	5747	0,748	0,912	0,841	0,849	0,792	0,826	0,748	0,913

2.6. Gradient Boosted

El flujo de trabajo de este algoritmo es:

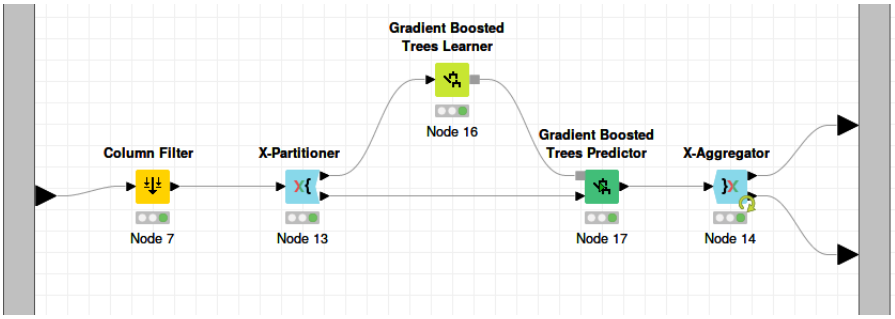


Figura 2.8: Workflow del algoritmo Gradient Boosted.

Comenzamos aplicando un *Column Filter* para eliminar valores no deseados para la

ejecución. Ejecutamos el predictor con un límite de 4 niveles de profundidad, 100 modelos y 0.1 de tasa de aprendizaje.

	TP	FP	TN	FN	TPR	TNR	PPV	Accuracy	F1-score	G-mean	Recall	AUC
Gradient Boosted	15662	3294	33282	7162	0,686	0,91	0,826	0,824	0,75	0,79	0,686	0,888

2.7. C4.5 (decision tree)

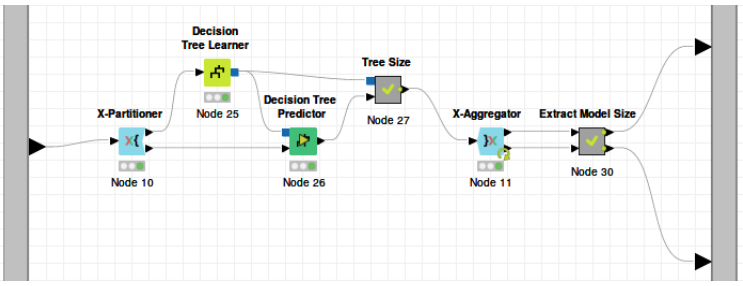


Figura 2.9: Workflow del algoritmo C4.5.

Este algoritmo no requiere ningún preprocesado, pues puede trabajar con valores perdidos y variables numéricas y nominales. Además, en este flujo de trabajo se han añadido metanodos para calcular el *model size*[1] del predictor ejecutado, es decir, el número medio de hojas en los árboles generados para cada partición. Los resultados son los siguientes:

	TP	FP	TN	FN	TPR	TNR	PPV	Accuracy	F1-score	G-mean	Recall	Model Size	AUC
Decision Tree	17526	5059	31195	5089	0,775	0,86	0,776	0,828	0,775	0,817	0,775	6381,8	0,841

3. Análisis de resultados

Veamos la tabla resumen de los valores de todos los algoritmos analizados (está ordenada por el valor de *accuracy* ascendentemente):

	TP	FP	TN	FN	TPR	TNR	PPV	Accuracy	F1-score	G-mean	Recall	AUC
Naive Bayes	14739	7170	29406	8085	0,646	0,804	0,673	0,743	0,659	0,721	0,646	0,794
Neural Network	13736	6135	30441	9088	0,602	0,832	0,691	0,744	0,643	0,708	0,602	0,793
Random Forest	11701	1912	34664	11123	0,513	0,948	0,86	0,781	0,642	0,697	0,513	0,855
Gradient Boosted	15662	3294	33282	7162	0,686	0,91	0,826	0,824	0,75	0,79	0,686	0,888
Decision Tree	17526	5059	31195	5089	0,775	0,86	0,776	0,828	0,775	0,817	0,775	0,841
KNN	7085	2003	16039	2631	0,729	0,889	0,78	0,833	0,754	0,805	0,729	0,543
Bagging	17077	3236	33340	5747	0,748	0,912	0,841	0,849	0,792	0,826	0,748	0,913

A continuación, hablemos de las diferentes medidas almacenados en la tabla.

[1]Solo se calculará el *model size* para C4.5 en este apartado; se volverá a tratar cuando hablemos del overfitting.

- **TP**, **FP**, **TN**, **FN**: True Positive, False Positive, True Negative y False Negative; su valor determina el número de aciertos y fallos del clasificador sobre la clase positiva y las clases negativas. En nuestro caso, la clase positiva es *non functional*, por lo que el valor de True Positive significará la cantidad de veces que el clasificador ha predicho que la instancia era *non functional* y ha acertado. El valor de True Negative significará la cantidad de veces que el clasificador ha predicho bien las clases negativas (*non functional* y *functional needs repair*). Por otra parte, False Positive y False Negative determinarán el número de veces que el predictor ha fallado en las clases positiva y negativas.

  Estos valores aparecen en la matriz de confusión, una matriz $n \times n$, donde n es el número de clases, que en la posición ij contiene la cantidad de veces que se ha predicho i siendo el valor real j.

  TP, FP, TN y FN los utilizaremos para el cálculo del resto de valores de la tabla.

- **TPR**, **TNR**: también llamados *sensitivity* y *specitifity*, dichos valores se obtienen como el cociente de True Positive entre el número total de instancias de clase positiva ($TP + FN$), y True Negative entre el número total de instancias de clase negativa ($TN + FP$), respectivamente. Veamos alguna gráfica para comparar los valores en función de los distintos algoritmos.

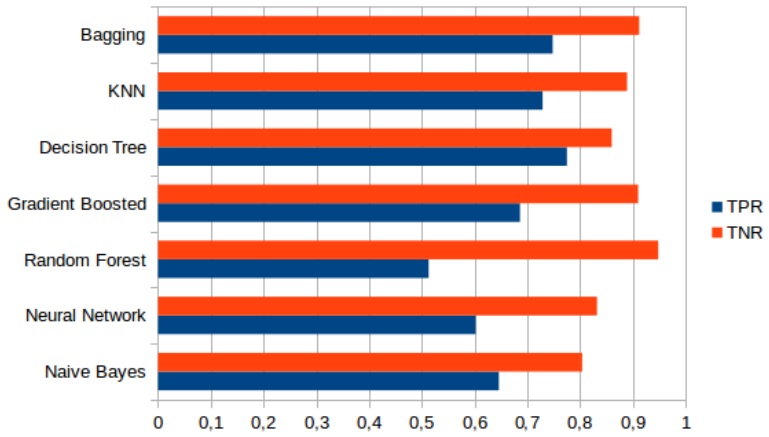


Figura 3.1: Comparación de *sesitivity* y *specitivity*.

Por norma, todos los algoritmos obtienen un mayor valor para TNR, lo que es normal ya que la clase negativa es mayor que la positiva para nuestro dataset. Además se da el caso de que en Random Forest es donde mayor valor de TNR se obtiene, pero también donde menor TPR hemos conseguido. Puede ser debido a que el modelo se ha centrado en la clase negativa a la hora de aumentar los casos de acierto, teniendo así un sobreaprendizaje sobre la clase negativa que conlleva un valor menor en la clase positiva.

La diferencia entre TPR y TNR en los diferentes algoritmos que podemos ver en la gráfica se verá reflejada más adelante en G-mean.

- **PPV**: también llamado *precision*, lo calcularemos como el cociente del número de predicciones positivas correctas (TP) entre el número total de predicciones positivas.

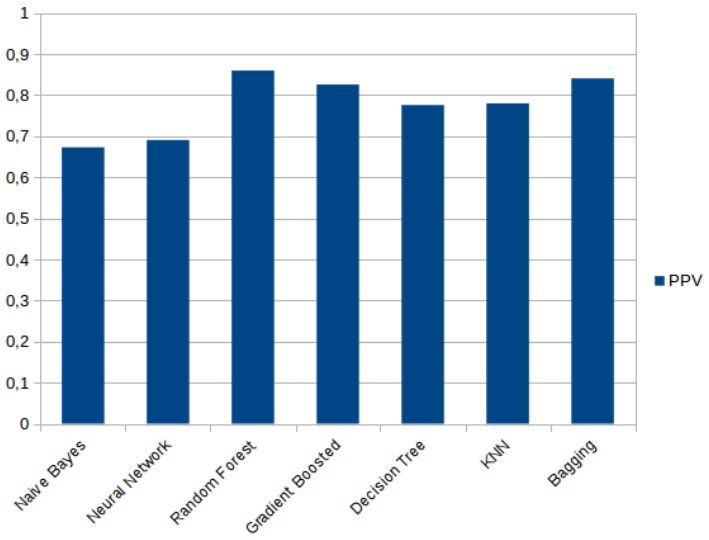


Figura 3.2: Positive Predictive Value de los resultados obtenidos.

Naive Bayes, al igual que en el resto de medidas tomadas, es el que menor valor obtiene. Por otra parte, Random Forest, aún habiendo obtenido el tercer peor valor de *accuracy*, es el mejor para esta medida, que puede ser debido al sobreaprendizaje comentado en el anterior punto.

- **Accuracy**: ya hemos comentado en la introducción cómo calcular este valor, utilizando las predicciones correctas entre el número total de predicciones.

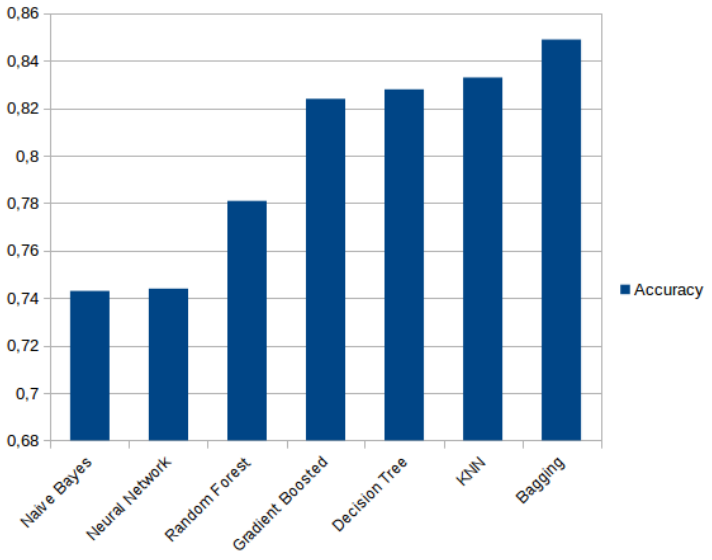


Figura 3.3: Accuracy de los resultados obtenidos.

Los algoritmos que han obtenido mejores resultados aquí han sido C4.5, K-NN y Bagging. Ya que K-NN obvia las instancias con valores perdidos el número total de predicciones realizadas es menor, lo que conlleva que ninguno de los elementos del dataset que se han clasificado tiene valores perdidos por lo que el clasificador funciona mejor sobre este conjunto de datos, y de ahí su buen resultado de *accuracy*. Respecto a C4.5 y Bagging, ya hemos comprobado con las anteriores gráficas que los árboles de decisión funcionan muy bien sobre nuestro dataset, y como Bagging explota este tipo de aprendizaje, conseguimos un buen modelo de predicción utilizando este algoritmo.

Aún así, estos resultados no deben ser concluyentes para determinar el mejor algoritmo ya que nuestras clases están desbalanceadas y convendrá tener en cuenta el resto de medidas de evaluación para discernir entre cuál es el mejor predictor para nuestro conjunto de datos.

- **F1-score** y **G-mean**: la medida F1-score será calculada como $F1 = 2 * \frac{PPV * TPR}{PPV + TPR}$, es decir, la media armónica entre PPV y TPR. Por otra parte, el cálculo de G-mean lo hemos comentado en la introducción y es la media geométrica de TPR y TNR.

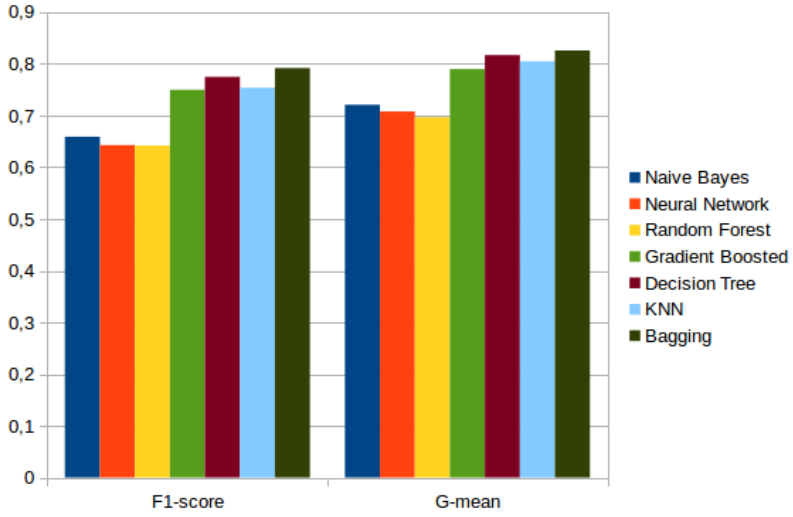


Figura 3.4: Gráfica de F1-score y G-mean.

Las medias armónica y geométrica discriminan los valores no balanceados; por tanto, en F1-score, cuanto más balanceados estén los valores de PPV y TPR (y más grandes sean) mayor será el valor de esta medida. El mismo comportamiento se dará en G-mean para TPR y TNR.

Por ejemplo, podemos ver en la gráfica que el valor de G-mean es mayor para C4.5 que para K-NN, y si comprobamos la diferencia entre sus valores de TPR y TNR vemos que para C4.5 $|TPR - TNR| = 0{,}085$ y para K-NN $|TPR - TNR| = 0{,}16$, teniendo así los valores de C4.5 más balanceados.

Random Forest obtendrá el peor resultado para estas medidas, pues $|TPR - TNR| = 0{,}435$ y $|PPV - TPR| = 0{,}347$.

- **AUC**: *area under curve*, es el valor del aréa que queda bajo la curva ROC, siendo 1 como máximo. Con esta medida podemos comprobar la capacidad que tiene nuestro clasificador de incrementar los True Positive a mayor velocidad que los False Positive.

	Naive Bayes	Neural Network	Random Forest	Gradient Boosted	Decision Tree	KNN	Bagging
AUC	0,794	0,793	0,855	0,888	0,841	0,543	0,913

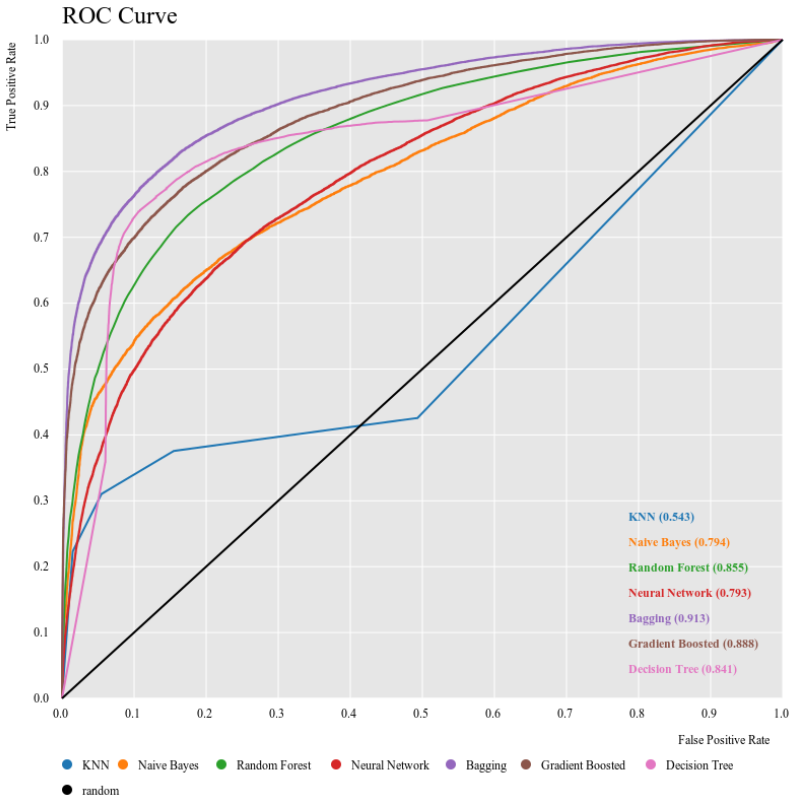


Figura 3.5: Curvas ROC de todos los algoritmos ejecutados.

K-NN obtiene un resultado muy malo, lo que es debido al hecho de que tenga tantos valores perdidos (y por tanto predicciones perdidas), pues la curva ROC ha sido calculada usando la probabilidad de que cada instancia sea de clase positiva usando la totalidad de los datos. De hecho, podemos comprobar como se genera una pequeña curva al principio en K-NN, para después pasar a ser un una función lineal.

Por otra parte, Bagging vuelve a ser el algoritmo con mejores resultados utilizando esta medida, lo que nos permite revalidar el hecho de que los algoritmos basados en árboles clasifican muy bien nuestro conjunto de datos.

4. Configuración de algoritmos

4.1. Rprop MLP

Hemos trabajado sobre este algoritmo modificando los diferentes parámetros que incluye el nodo de Knime: número máximo de iteraciones (por defecto 100), número de capas escondidas (por defecto 1) y número de neuronas por capa (por defecto 10).

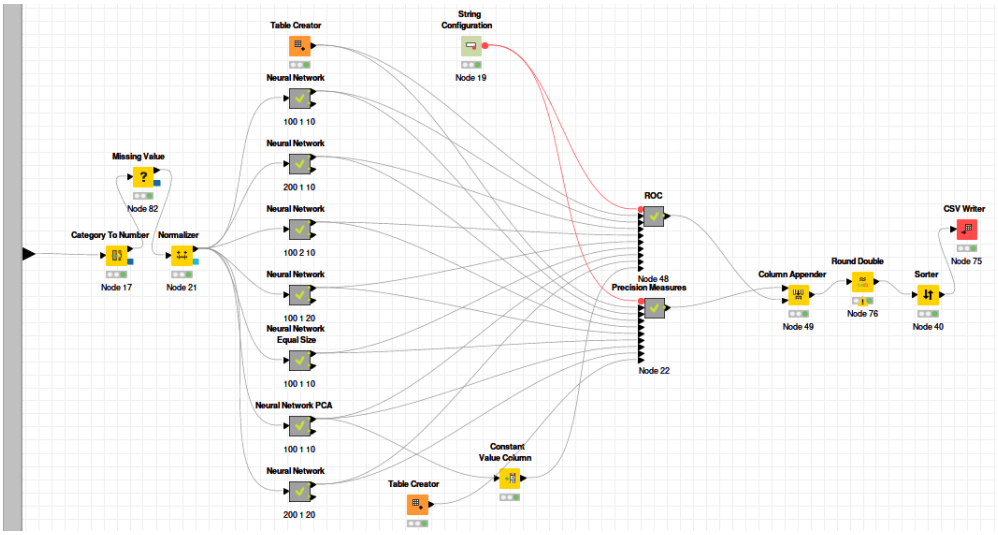


Figura 4.1: Workflow para la comparación entre las distintas configuraciones de MLP.

Como vemos, comenzamos convirtiendo todos los valores categóricos a numéricos para después imputar los valores perdidos y normalizar; este ligero preprocesado se aplicará a todas las distintas configuraciones del algoritmo. En cada metanodo se ejecutará un validación cruzada para el clasificador y el predictor Rprop MLP, modificando los distintos parámetros de configuración del clasificador. En el caso de los metanodos Neural Network Equal Size y Neural Network PCA, en el primero se ha ejecutado Rprop MLP modificando el dataset para que haya un número equitativo de clases entre las diferentes instancias (manteniendo el tamaño del dataset, 59400), y en el segundo se han eliminado diferentes características utilizando una matriz de covarianzas (lo comentaremos en el apartado de preprocesado).

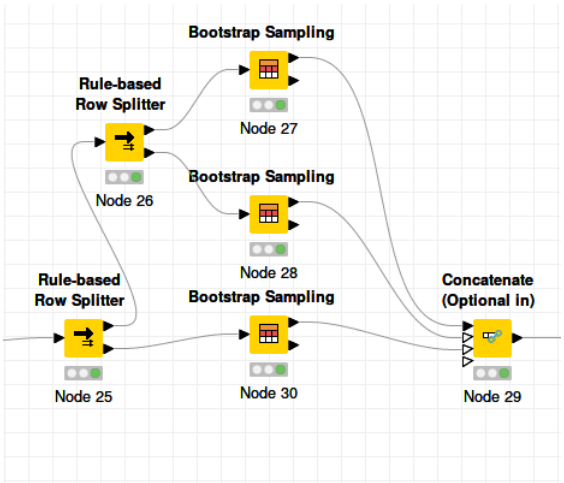


Figura 4.2: Procesado de los datos para que el dataset tenga las clases balanceadas.

En la imagen superior, el primer *Rule-based Row Splitter* separará las instancias de clase *functional* del resto, y en el segundo se separarán las *non functional* de las *functional needs repair*. Con el nodo *Bootstrap Sampling* seleccionamos el número de instancias que queremos generar para cada clase (19800) y finalmente concatenamos los datos obtenidos, teniendo así 59400 instancias con sus clases balanceadas.

Los resultados obtenidos por los distintos algoritmos, ordenados por el valor de *accuracy*, son los siguientes:

	TP	FP	TN	FN	TPR	TNR	PPV	Accuracy	F1-score	G-mean	AUC
MLP PCA	13147	6598	29978	9677	0,576	0,82	0,666	0,726	0,618	0,687	0,775
MLP equal size	11874	7689	31911	7926	0,6	0,806	0,607	0,737	0,603	0,695	0,774
MLP 100-1-10	13736	6135	30441	9088	0,602	0,832	0,691	0,744	0,643	0,708	0,793
MLP 100-2-10	12848	5043	31533	9976	0,563	0,862	0,718	0,747	0,631	0,697	0,794
MLP 100-1-20	13626	5512	31064	9198	0,597	0,849	0,712	0,752	0,649	0,712	0,8
MLP 200-1-10	14103	5446	31130	8721	0,618	0,851	0,721	0,761	0,666	0,725	0,814
MLP 200-1-20	14288	5045	31531	8536	0,626	0,862	0,739	0,771	0,678	0,735	0,824
MLP 200-2-20	14250	4439	32137	8574	0,624	0,879	0,762	0,781	0,687	0,741	0,829

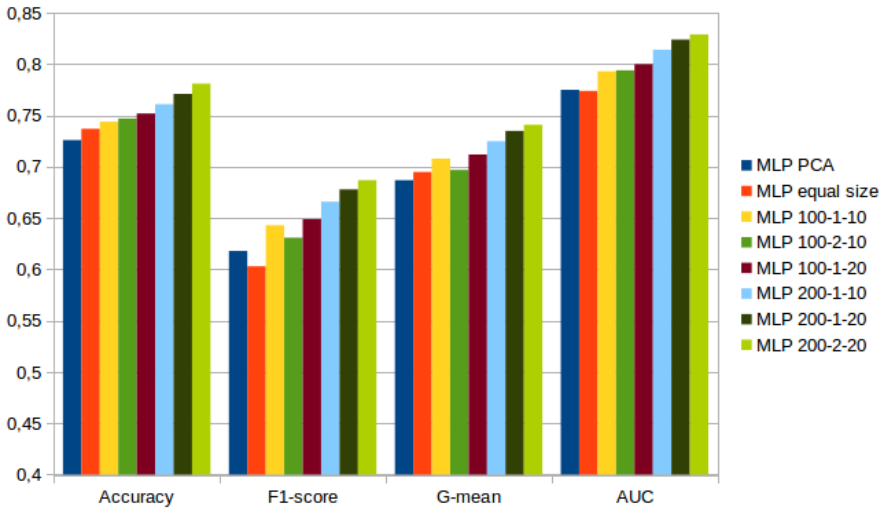


Figura 4.3: Gráfica con las medidas de comparación para las configuraciones de MLP.

Aunque MLP PCA y MLP equal size hayan sido ejecutados con los mismo parámetros que MLP 100-1-10, estos dos primeros obtienen un peor resultado. Esto puede ser debido a una eliminación errónea de categorías y a que el procesado de equal size que hemos realizado para obtener el mismo número de instancias de cada clase haya eliminado demasiadas instancias de la clase *functional* que fueran relevantes. Por tanto, habría que ser cautos a la hora de realizar estos preprocesados sobre nuestro dataset.

En contraste, vemos que cuantas más iteraciones, más capas ocultas y más nodos por capa añadimos obtenemos valores mayores para las medidas de comparación de nuestros

algoritmos; aunque también hay que notar que cuando aumentamos el número de capas y el número de nodos por capa es pequeño, el valor de F1-score y G-mean disminuye, teniendo así que TPR está menos balanceado con TNR y PPV.

Por último, habría que tener en cuenta también la capacidad de cómputo, así como el tiempo que tenemos para la clasificación de nuestros datos, pues cuanto mayor es el número de iteraciones, capas y nodos por capa, mayor tiempo de ejecución necesitaremos.

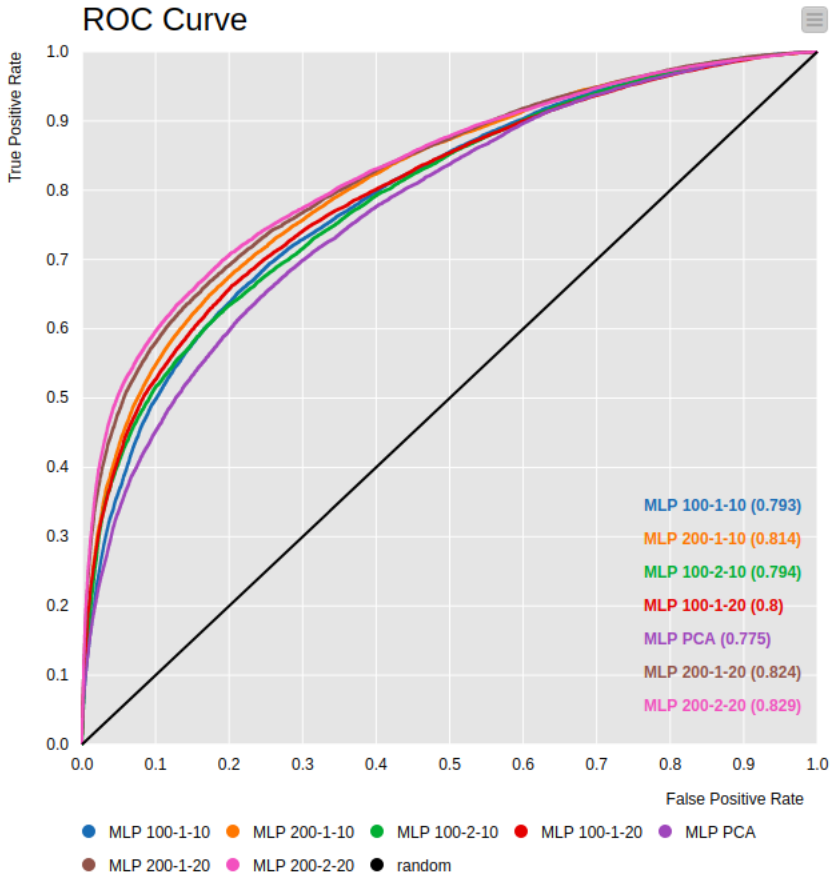


Figura 4.4: Curva ROC de los algoritmos MLP.

En las distintas curvas ROC podemos ver que los arcos que forman los distintos algoritmos siguen un patrón similar, aumentando el área que dejan bajo ello cuando aumentamos el número de iteraciones, capas y nodos por capa.

4.2. Random Forest

En este apartado comprobaremos si podemos solucionar el *overfitting* de Random Forest modificando su configuración[2].

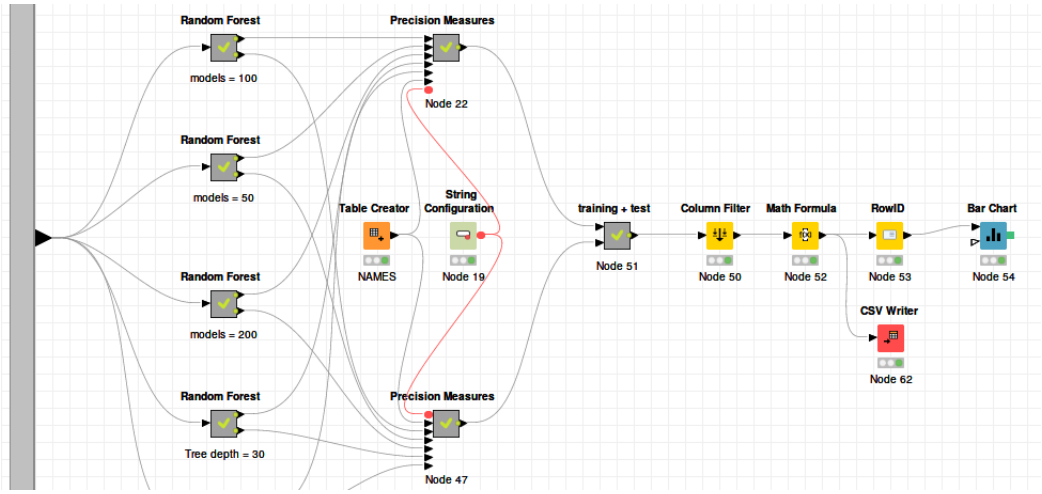
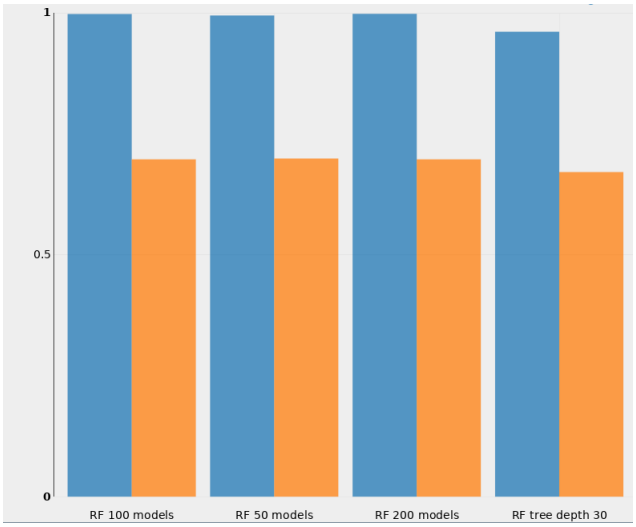


Figura 4.5: Workflow para calcular el overfitting de Random Forest.
La estructura del flujo de trabajo para comprobar el overfitting es la descrita en el tutorial de Youtube del profesor Jorge Casillas.

Los parámetros que hemos modificado han sido el número de modelos generados y la profundidad de dichos árboles. Obtendremos los siguientes valores:



[2]La imagen está cortada porque todavía no mostramos un nodo que nos va a ayudar a mejorar el overfitting, en el apartado de contenido adicional.

	G-mean (tra)	G-mean (tst)	overfitting
RF 100 models	0,997	0,697	1,431
RF 50 models	0,994	0,699	1,423
RF 200 models	0,998	0,697	1,431
RF tree depth 30	0,961	0,671	1,432

Podemos ver que Random Forest es un modelo de clasificación que tiende al sobreaprendizaje y que la configuración de sus parámetros no soluciona el problema del overfitting, manteniendo un valor estable en torno a 1,4. El modelo con un valor de sobreaprendizaje más cercano a 1 es RF 50 models, aunque también hay que tener en cuenta que en este caso porcentaje de acierto para el conjunto de entrenamiento disminuye más de lo que aumenta el valor del conjunto test. Por otra parte, vemos que aumentar la profundidad de los árboles (por defecto a 30) no soluciona el sobreaprendizaje y además disminuye el porcentaje de acierto en training y test.

En la sección "Contenido Adicional" intentaremos solucionar el sobreaprendizaje con otros métodos.

5. Procesado de datos

En este apartado realizaremos un preprocesado de los datos para intentar mejorar Naive Bayes, MLP, Random Forest y C4.5. Para todos los algoritmos eliminaremos el atributo con más valores perdidos (scheme_name con casi más de un 50 % de valores perdidos) e imputaremos los valores perdidos en el resto de atributos y finalmente realizaremos un preprocesado algo diferente para cada uno de ellos.

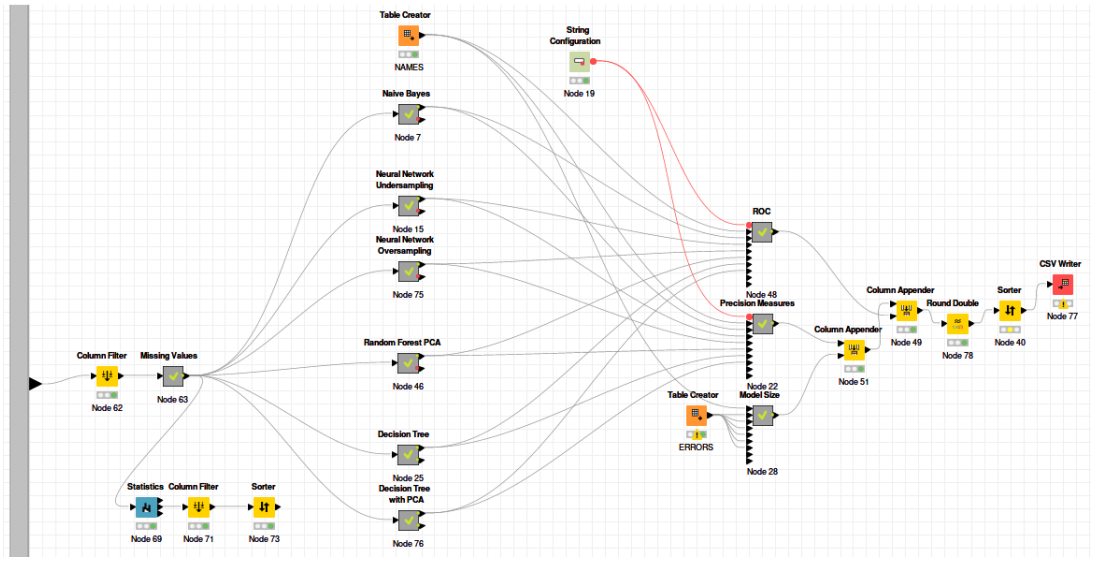


Figura 5.1: Workflow para los algoritmos con preprocesado.

El nodo *Missing Value* de Knime solo trabaja imputando valores teniendo en cuenta diferentes estadísticos. Por tanto, se ha creado el metanodo Missing Value que se encargará de determinar los valores perdidos mediante K-NN, obteniendo así mejores resultados.

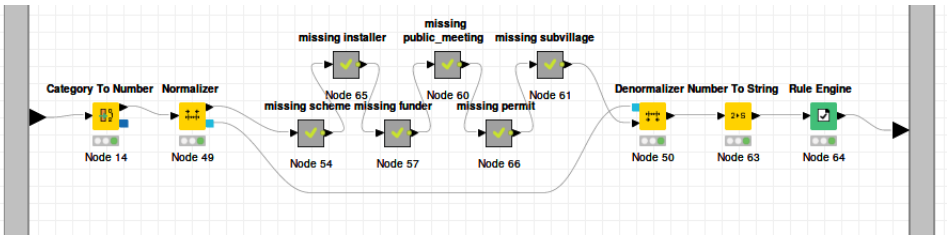


Figura 5.2: Metanodo Missing Value.

Comenzaremos pasando todos los atributos a numéricos (incluida la clase) y normalizándolos, pues K-NN necesita este preprocesado para funcionar correctamente. Tras ello se ejecutará un metanodo para cada atributo con valores perdidos que aplicará K-NN tomando como clase dicho atributo.

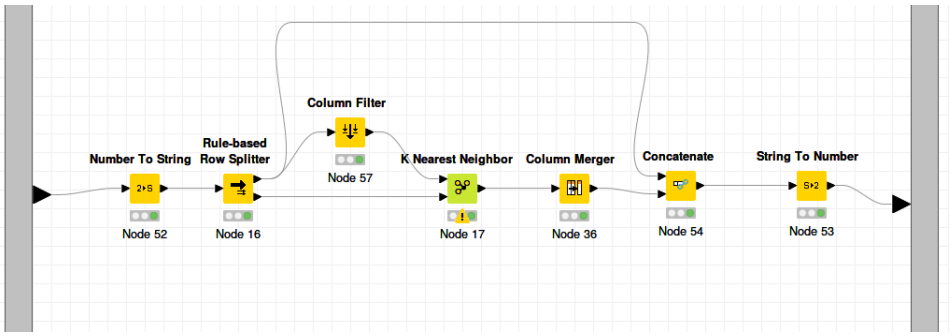


Figura 5.3: Metanodo para aplicar K-NN a un atributo con valores perdidos.

missing El metanodo definido para todas las columnas con valores perdidos será similar al de arriba. Comenzaremos pasando el valor que queremos inferir a String para que K-NN lo reconozca como atributo de clase. Tras ello, separaremos en training y test; training tendrá eliminadas las columnas con valores perdidos y test contendrá todas las instancias que no tienen definido valor que queremos determinar. A continuación ejecutaremos K-NN y modificaremos la columna con los valores perdidos por los valores que acabamos de determinar, para después unir con los datos de entrenamiento. Finalmente volveremos a poner el atributo que queríamos determinar como valor numérico y pasaremos al siguiente metanodo, que hará lo mismo con la siguiente columna con valores perdidos.

Una vez inferidos todos los valores perdidos, desnormalizaremos los datos y volveremos a dar su nombre original a las distintas clases (*functional, non functional* y *function needs repair*).

Pasaremos un nodo *Missing Value* de Knime en cada algoritmo por si algún valor se hubiera quedado sin asignar mediante K-NN.

5.1. Atributos no relevantes

Esto lo hemos usado en Naive Bayes, MLP y C4.5. Nos basaremos en una matriz de covarianzas entre los distintos atributos de nuestro dataset para determinar qué características son irrelevantes y podemos eliminar. Para ello, simplemente tendremos que utilizar los nodos *Linear Correlation* y *Correlation Filter*; el primero de ellos generará la matriz de covarianzas que vemos a continuación:

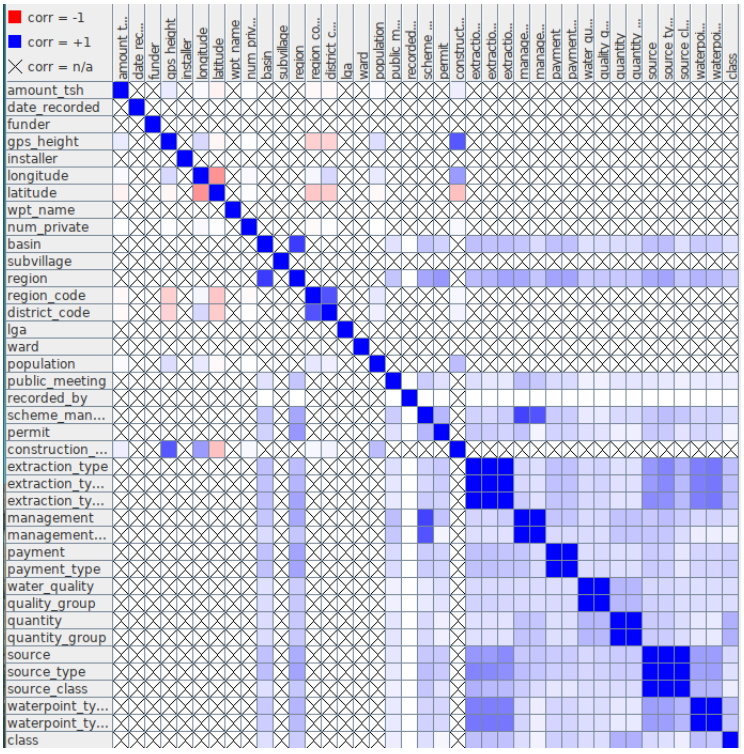


Figura 5.4: Matriz de covarianzas de nuestro dataset.

Cuanto más azul, más cercano a 1 y cuanto mas rojo más cercano a -1. Con el nodo *Correlation Filter* estableceremos a partir de qué valor de covarianza eliminaremos un atributo, teniendo que cuanto menor sea este umbral, menos características quedarán en nuestro dataset. En nuestro caso, con un umbral de 0.5, se eliminarán 12 atributos.

5.2. Naive Bayes

Para este algoritmo hemos aplicado la matriz de covarianzas para obtener un menor número de características y hemos utilizado un *Auto-Binner*, que almacena los atributos en varios *contenedores* (en concreto en 50) para la ejecución de nuestro algoritmo. Esto es una buena opción de preprocesado para el algoritmo de Naive Bayes, pues en su configuración hay que elegir el máximo número de valores nominales únicos por atributo (que también valdrá 50).

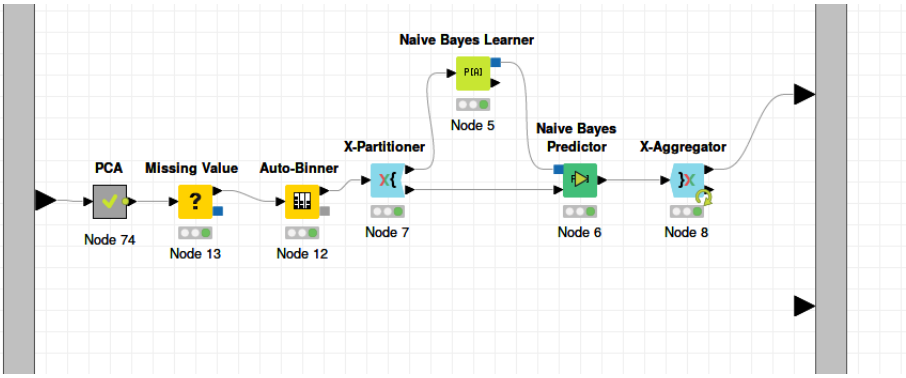


Figura 5.5: Workflow de Naive Bayes con preprocesado.

Los resultados que hemos obtenido en la ejecución de este algoritmo, comparados con el algoritmo sin preprocesado, son los siguientes:

	TP	FP	TN	**FN**	**TPR**	**TNR**	**PPV**	**Accuracy**	**F1-score**	**G-mean**	**AUC**
NB	14739	7170	29406	8085	0,646	0,804	0,673	0,743	0,659	0,721	0,794
NB Preprocesado	14504	7790	28786	8320	0,635	0,787	0,651	0,729	0,643	0,707	0,782

Con este procesado no obtenemos mejores medidas sobre nuestro dataset, es más disminuyen un poco, lo que puede ser debido a que con Auto-Binner hemos añadido diferentes valores numéricos que pudieran ser relevantes a un *bin*, perdiendo así su valor real.

5.3. Rprop MLP

Para este algoritmo también hemos eliminado características no relevantes, y además hemos modificado el tamaño de nuestro dataset mediante *Oversampling* y *Undersampling* para que nuestras etiquetas se mantengan balanceadas.

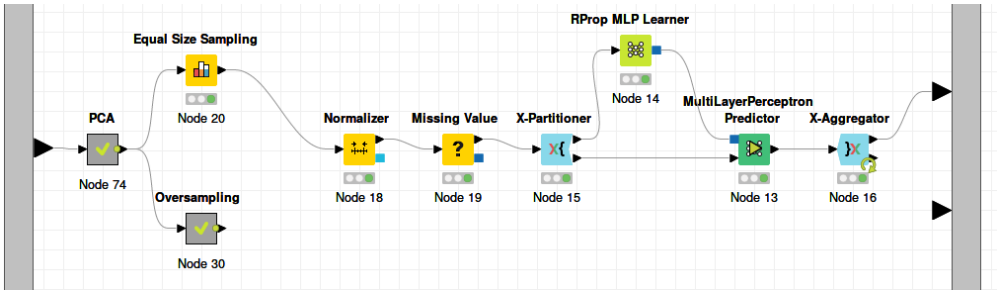


Figura 5.6: Workflow para MLP usando preprocesado y Undersampling.

Para el caso de Undersampling, el nodo *Equal Size Sampling* eliminará instancias de la clase *functional* y *non functional* para que estén balanceadas con *functional needs repair*. Por otra parte, para Oversampling utilizaremos el nodo *Bootstrap Sampling* para obtener nuevas instancias (generadas por copia de otras ya existentes) de *functional needs repair* y *non functional*, teniendo así en todas ellas el mismo número de instancias que las clasificadas con *functional*.

	TP	FP	TN	FN	TPR	TNR	PPV	Accuracy	F1-score	G-mean	AUC
MLP	13736	6135	30441	9088	0,602	0,832	0,691	0,744	0,643	0,708	0,793
MLP Undersampling	2435	1325	7309	1882	0,564	0,847	0,648	0,752	0,603	0,691	0,779
MLP Oversampling	19076	9492	55026	13183	0,591	0,853	0,668	0,766	0,627	0,71	0,8

Comprobamos con los resultados que, balanceando las clases de nuestro conjunto de datos, obtenemos mejores resultados que si trabajamos con un dataset desbalanceado. Además, Oversampling funciona mejor que Undersampling, pues con este segundo se puede dar el caso de que eliminemos instancias de otras clases que sean muy relevantes para el aprendizaje de nuestro clasificador, mientras que Oversampling mantiene todas las instancias originales y genera nuevas para las clases desbalanceadas.

También podemos ver que los valores de F1-score y G-mean son menores en el caso de las clases balanceadas, lo que es normal ya que antes la clase negativa estaba formada por un 38 % de las instancias frente al 62 % de la clase positiva, y ahora habrá un tercio de instancias en la clase negativa y dos tercios en la clase positiva, y por tanto el valor de TPR diferirá mas con el de PPV y TNR. Aún así, la mejora de los resultados con usando Oversampling hace que podamos obtener un mejor valor de G-mean.

5.4. C4.5

Al igual que en el caso anterior, hemos ejecutado dos veces el clasificador, uno eliminando variables inútiles y otro sin hacerlo. Aparte, hemos aplicado Oversampling al conjunto de datos para la ejecución de los dos algoritmos.

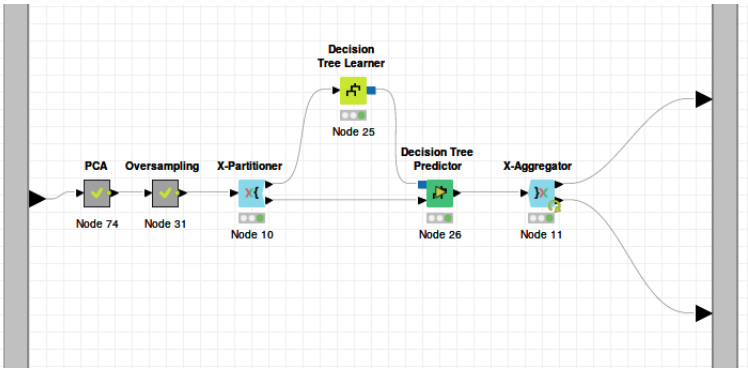


Figura 5.7: Workflow de C4.5 con preprocesado y eliminando variables inútiles.

	TP	FP	TN	FN	TPR	TNR	PPV	Accuracy	F1-score	G-mean	AUC
C4.5	17526	5059	31195	5089	0,775	0,86	0,776	0,828	0,775	0,817	0,841
PCA C4.5 pre	28478	4539	59979	3781	0,883	0,93	0,863	0,914	0,873	0,906	0,929
C4.5 pre	28520	4359	60159	3739	0,884	0,932	0,867	0,916	0,876	0,908	0,931

Podemos comprobar que, como era de esperar, el preprocesado mejora (y bastante) los resultados de C4.5 sobre nuestro dataset, siendo el modelo en el que eliminamos características algo peor que en el que no eliminamos, aunque la diferencia es mínima.

Con esto hemos visto que balancear las clases es una buena opción a la hora de realizar un preprocesado de los datos, y siempre será mejor optar por el oversampling para no eliminar posibles instancias relevantes dentro de nuestro dataset.

6. Interpretación de resultados

Para extraer conclusiones sobre los factores que determinan la clase podemos reutilizar la matriz de covarianzas que hemos mostrado en el anterior punto:

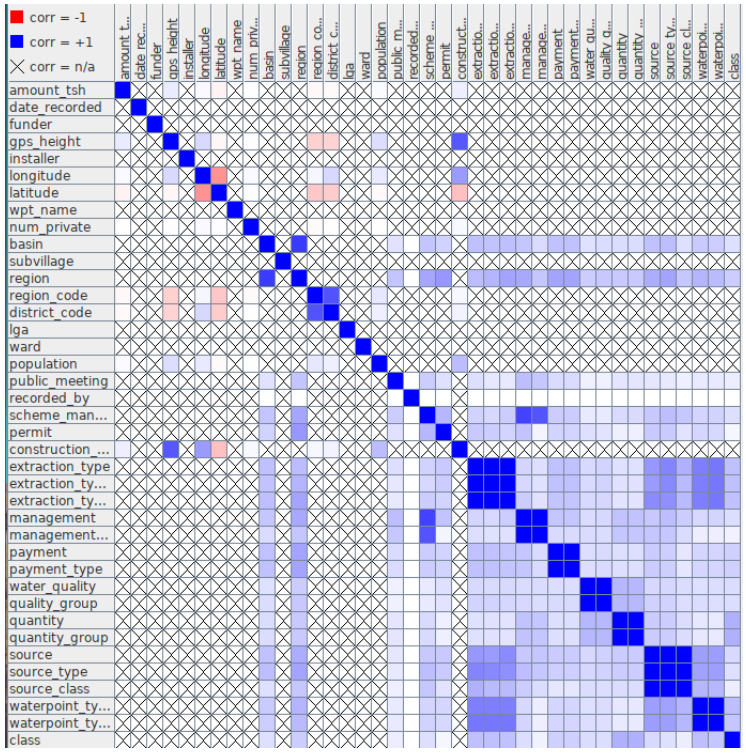


Figura 6.1: Matriz de covarianzas de nuestro dataset.

En esta matriz simétrica se representa la relación que hay entre los distintos atributos, por lo que si nos fijamos en la última columna, donde aparece el valor de covarianza de la clase, podremos determinar cuál de nuestras características es más relevante a la hora de tomar una decisión.

Los valores con un azul más intenso son los más relevantes con el atributo que estemos estudiando (por eso los elementos de la diagonal tienen el azul más oscuro, ya que un elemento está totalmente relacionado consigo mismo). Si visionamos esta matriz en Knime, pasando el ratón por encima de cada casilla podremos obtener el valor de la covarianza. Así, las características más relacionadas con la clase son: region (0.2), extraction_type (0.249), quantity (0.3092) y waterpoint_type (0.25). Era de esperar que esto valores fueran relevantes, ya que tratan la cantidad de agua así como el método de extracción y la zona donde se encuentra el pozo, aunque también se esperaba una relación alta entre la clase y el atributo water_quality, pues la calidad del agua podría determinar más o menos averías en la bomba de extracción.

Siguiendo con la extracción de conclusiones, veamos los primeros niveles de la primera iteración de cross validation sobre el árbol que mejores resultados nos ha dado sobre nuestro dataset, C4.5 con oversampling.

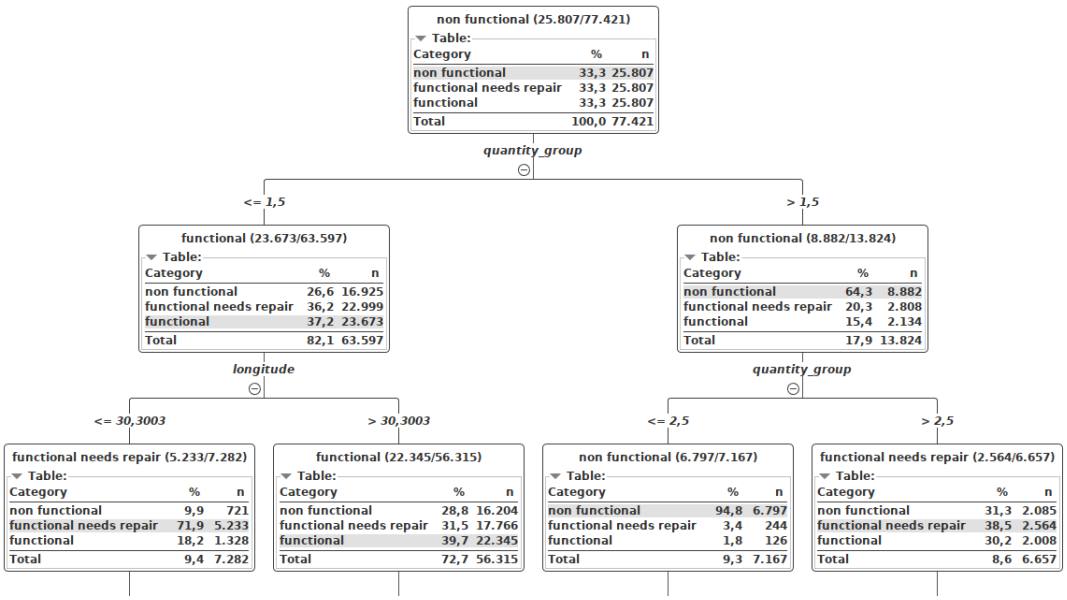


Figura 6.2: Primeros niveles de un árbol de decisión sobre nuestro dataset.

Vemos que en el primer paso comprueba el atributo quantity_group para determinar en qué grupo se encuentra la instancia a clasificar. Hay que tener en cuenta que quantity y quantity_group están muy relacionados, por lo que si eliminaramos las características inútiles lo más probable es que se eliminar quantity_group y lo primero que comprobara el árbol fuera el atributo quantity. A continuación, si nos vamos por la rama de la derecha se volverá a comprobar el grupo, pero si nos vamos por la izquierda se comprueba el valor de longitude, es decir, la distancia del pozo al ecuador de la Tierra. Tiene sentido que se compruebe esto ya que cuanto más cerca se esté del ecuador mayores serán las temperaturas y por tanto el agua se evaporará y la cantidad en el pozo será menor, lo cuál nos lleva de nuevo a ver la razón de por qué el árbol ha elegido como primera decisión determinar en qué grupo de cantidad de agua se encuentra la instancia a clasificar.

Yendo por la rama derecha del árbol podemos ver que cuando el quantity_group sea mayor que 1.5 y menor que 2.5, es decir, cuando valga 2, lo más probable es que la clase sea *non functional*. Cojamos algunos datos para comprobarlo:

D quantity_group	S class
2	non functional
2	non functional
2	non functional
2	non functional
1	non functional
2	non functional
2	non functional
2	non functional
2	non functional
2	non functional

Figura 6.3: Algunos valores de quantity_group junto a su clase.

Como podemos ver, en todas las instancias el valor de quantity_group es 2, salvo en una de ellas que falla, lo que era de esperar ya que según el árbol había un 94 % de acierto con esa regla.

7. Contenido adicional

Para esta sección hemos querido intentar reducir el sobreaprendizaje del algoritmo Random Forest, comentado en el apartado de configuración de algoritmos. Para ello hemos aplicado el algoritmo con sus valores por defecto, es decir, generando 100 modelos, pero previamente eliminado las variables inútiles de nuestro dataset; comprobaremos si ello consigue un mejor resultado.

A continuación se muestran las G-mean de los distintos algoritmos y su valor de Overfitting:

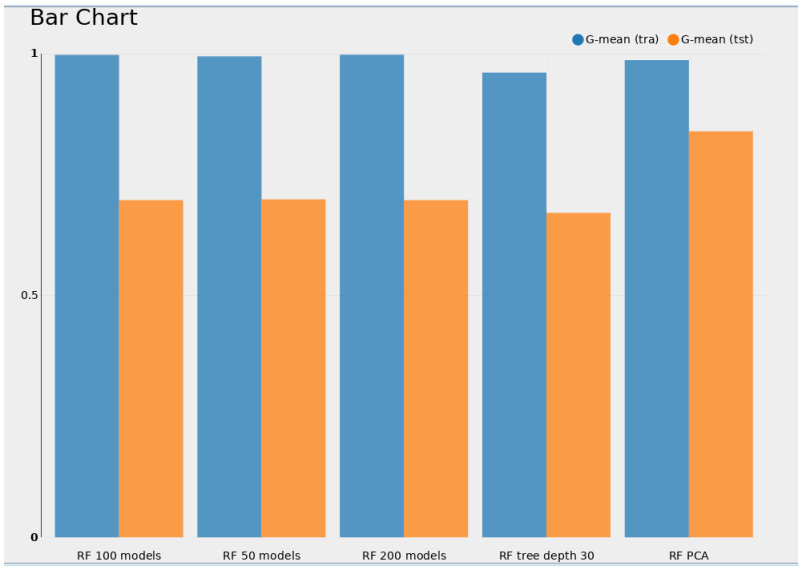


Figura 7.1: Overfitting del algoritmo Random Forest junto a una mejora.

	G-mean (tra)	G-mean (tst)	overfitting
RF 100 models	0,997	0,697	1,431
RF 50 models	0,994	0,699	1,423
RF 200 models	0,998	0,697	1,431
RF tree depth 30	0,961	0,671	1,432
RF PCA	0,986	0,839	1,175

Podemos ver que hemos obtenido una mejora significativa respecto a las anteriores configuraciones de Random Forest, consiguiendo disminuir 0.25 puntos el sobreaprendizaje de

nuestro modelo. Esto es debido a que las ramas de los árboles que anteriormente Random Forest estaba explotando, causando el sobreaprendizaje, han sido podadas al eliminar las distintas variables inútiles. Cuando mantenemos estos atributos, Random Forest continúa entrenando hasta que encuentra una solución para esas variables, las cuales podemos considerar soluciones sin valor. Al haber eliminado las soluciones sin valor, el clasificador se puede centrar más en explotar la precisión sobre el conjunto de validación.

Si nos fijamos en el G-mean del training, el valor de RF PCA ha disminuido respecto a los anteriores (obviando RF tree depth 30 que ya comentamos que daba malos resultados). Esta disminución se puede soportar, ya que es insignificante en comparación con la gran mejora que se consigue en el conjunto test.

8. Bibliografía

Transparencias de clase de teoría y prácticas junto a los tutoriales de Youtube subidos por Jorge Casillas.