Práctica 1:
Análisis Predictivo Empresarial Mediante Clasificación

Curso 2019/2020

SOFÍA ALMEIDA BRUNO
sofialmeida@correo.ugr.es

Grupo IN 2
Jueves 9:30-10:30

Índice

1. Introducción **2**

2. Resultados obtenidos **3**
2.1. ZeroR . . . 3
2.2. Árbol de decisión . . . 4
2.3. k-NN . . . 6
2.4. Red neuronal . . . 7
2.5. Naive Bayes . . . 8
2.6. Random Forest . . . 9
2.7. Boosting . . . 10

3. Análisis de resultados **11**

4. Configuración de algoritmos **11**

5. Procesado de datos **11**

6. Interpretación de resultados **11**

7. Contenido adicional **11**

8. Bibliografía **11**

1. Introducción

En este práctica se abordará un problema de clasificación del mundo real para, mediante el uso de los algoritmos de clasificación supervisada vistos en clase de teoría y las herramientas y recursos expuestos en clase de prácticas, realizar una predicción sobre el mismo y analizar cómo de buena es esta clasificación. Se compararán distintos algoritmos y se examinará la predicción obtenida en función a los mismos según distintos criterios de precisión.

El problema con el que se trabajará proviene de la plataforma "Driven data", usa los datos de "Taarifa" (API web libre que está trabajando en un poryecto de innovación en Tanzania) y del Ministerio de Agua de Tanzania. El objetivo es predecir qué bombas de agua funcionan, cuáles necesitan alguna reparación y cuáles están rotas. Es decir, estamos ante un problema de clasificación con tres clases diferentes. Se trata de predecir mediante variables como: qué tipo de bomba es, cuándo se instaló, cantidad de agua disponible,... ante qué tipo de bomba de agua nos encontramos. Saber qué puntos de agua fallarán permitirá mejorar las tareas de mantenimiento y asegurar que hay agua limpia y potable disponible para las comunidades de Tanzania.

Abordaremos el problema a partir de un conjunto de datos formado por 59400 instancias, de las cuales conocemos información sobre 39 variables, además de su clase (una de las tres ya mencionadas). En primer lugar, usando el nodo `Pie/Donut Chart`, observamos en la Figura 1 la frecuencia de las clases: de todas las instancias 32259 son bombas de agua funcionales, 22824 no funcionales y 4317 funcionales pero necesitan una reparación.

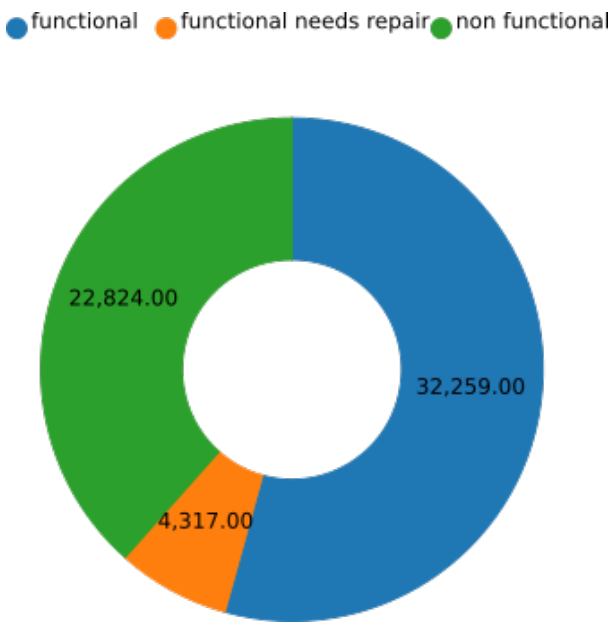

Figura 1: Número de instancias de cada clase

Las clases están desbalanceadas, observamos una gran diferencia en el número de ejemplos de bombas funcionales y aquellas que pese a ser funcionales requieren mantenimiento. En la Figura 2 podemos ver que más de la mitad de las instancias son bombas de agua funcionales (un 54 % de ellas), las no funcionales ocupan un 38 % de las instancias y las funcionales que necesitan reparación forman la clase minoritaria, con tan solo un 7 % de los ejemplos.

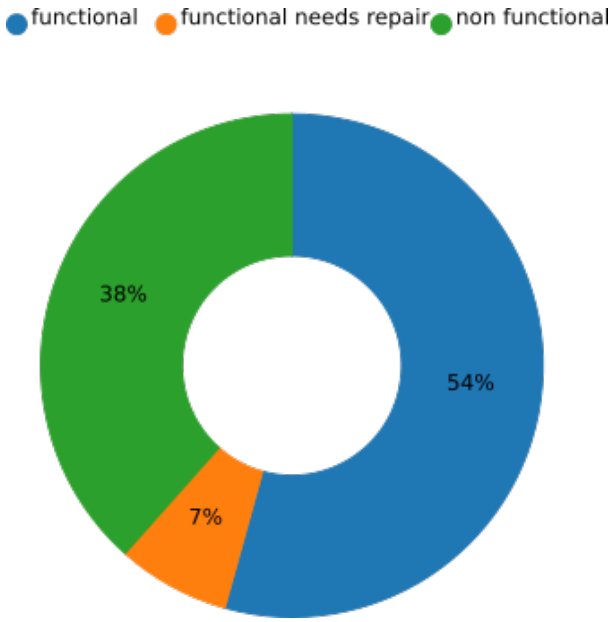

Figura 2: Porcentaje de instancias de cada clase

Consideraremos como clase positiva "non functional", ya que queremos predecir cuáles son las bombas que no funcionan para poder sustituirlas.

Nada más cargar el fichero observamos que es un conjunto de datos que posee valores perdidos, además de algunos valores "unknown".

Toda la experimentación se realizará usando una validación cruzada de 5 particiones. La semilla aleatoria empleada en aquellos algoritmos que lo requieran será: 12345. Los experimentos realizados en esta práctica se ejecutaron en un ordenador con sistema operativo Ubuntu 16.04 con procesador Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz.

Se utilizará la validación cruzada en la ejecución de todos los algoritmos, mediante los nodos `X-Partitioner` y `X-Aggregator` de KNIME. Se configuran para crear 5 particiones, luego en cada experimento se utilizará un conjunto de entrenamiento de tamaño 47520 y un conjunto de prueba formado por 11880 instancias.

Hemos visto en clase que comparar los algoritmos solo por la precisión que consiguen en la predicción no es suficiente, ya que en conjuntos desbalanceados malos algoritmos podrían obtener una alta precisión. Así, utilizaremos medidas sensibles al desbalanceo. Se siguió el tutorial proporcionado por el profesor de prácticas sobre cómo comparar diferentes algoritmos para obtener las tablas de resultados.

2. Resultados obtenidos

2.1. ZeroR

Para comenzar (y sin incluirlo como algoritmo a estudiar), he decidido observar el comportamiento del clasificador ZeroR. Este clasificador predice que cualquier instancia pertenecerá a la clase mayoritaria. Aunque ya sabemos que no obtendremos un buen resultado utilizando este clasificador porque solo clasificará correctamente las instancias que verdaderamente pertenezcan a la clase mayoritaria, nos servirá para tener una cota inferior de las medidas. Si en algún momento durante el desarrollo de la práctica obtuvieramos resultados peores que los obtenidos con este clasificador sospecharemos que estamos haciendo algo mal.

Podemos observar el metanodo creado en KNIME para este algoritmo en la Figura 3. Se ha utilizado el nodo `ZeroR` de `Weka`.

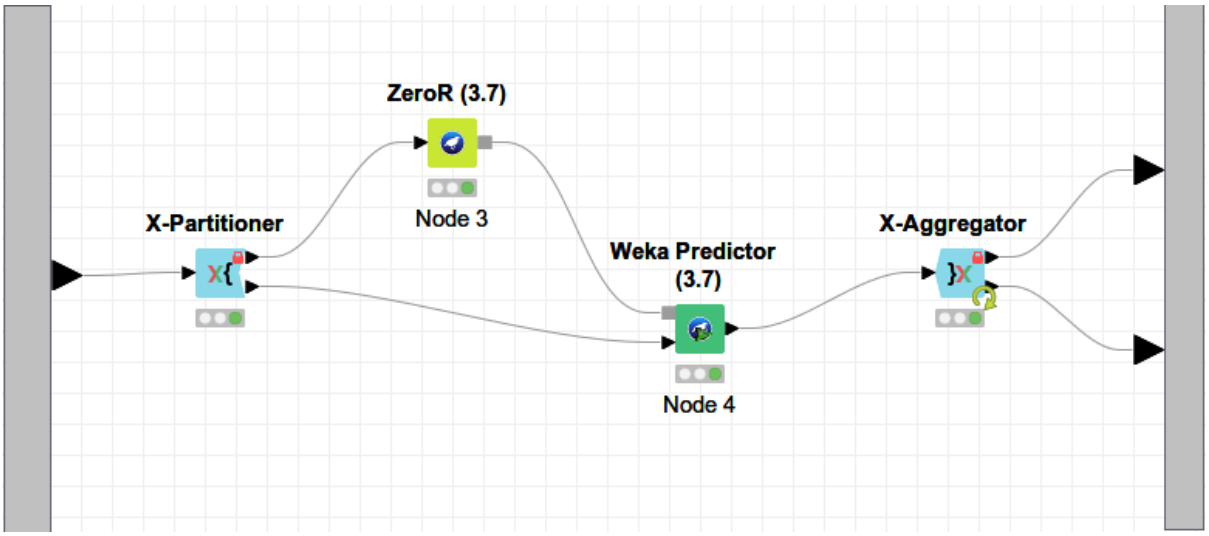

Figura 3: Metanodo ZeroR

Utilizamos un nodo `Scorer` para conocer su matriz de confusión, que encontramos en la Tabla 1, y, tal como debería, todas las instancias son clasificadas como funcionales.

Tabla 1: ZeroR - Matriz de confusión

	functional	non functional	functional needs repair
functional	32259	0	0
non functional	22824	0	0
functional needs repair	4317	0	0

En la Tabla 2 se encuentran las medidas de precisión obtenidas con este algoritmo. En este caso, conociendo la distribución de las clases, se podrían haber calculado manualmente sin necesidad de ejecutar el algoritmo. Sabemos que todas las instancias serán clasificadas como funcionales y nuestra clase positiva es no funcional, luego no habrá verdaderos positivos (tampoco falsos positivos).

Tabla 2: ZeroR - criterios de precisión

	TP	FP	TN	FN	TPR	TNR	PPV	Accur.	F1-score	G-mean
ZeroR	0	0	36576	22824	0	1	?	0.61576	?	0
C4.5 - gini	17456	5016	31276	5158	0.77191	0.86179	0.77679	0.82728	0.77434	0.81561
C4.5 gain	16536	4730	31658	6177	0.72804	0.87001	0.77758	0.81545	0.752	0.79587

2.2. Árbol de decisión

El primer algoritmo elegido es uno basado en árboles de decisión. Estos algoritmos parten de todos los ejemplos y van seleccionando atributos para dividir el conjunto de ejemplos en función al valor de estos atributos. Como criterio para seleccionar las variables se usa el índice Gini, así que nos encontramos con un algoritmo CART.

El índice Gini mide con qué frecuencia un elemento elegido de forma aleatoria de un conjunto sería etiquetado incorrectamente si se etiqueta aleatoriamente de acuerdo a la distribución de clases en el

subconjunto. Dado un conjunto de datos T con ejemplos pertenecientes a n clases, el índice de Gini se define como:

$$gini(T) = 1 - \sum_{j=1}^{n} p_j^2,$$

donde p_j es la frecuencia relativa de la clase j en T. Este índice valdrá 0 cuando todos los ejemplos de un nodo sean de la misma clase.

Lo ejecutaremos en KNIME mediante ...

Obtenemos la matriz de confusión mostrada en la Tabla 3.

Tabla 3: CART - Matriz de confusión

	functional	non functional	functional needs repair
functional	26399	4265	1351
non functional	4644	17456	514
funcional needs repair	2075	751	1451

Hemos acudido al nodo `Scorer` para obtener esta matriz y nos damos cuenta de que tiene un triángulo a modo de advertencia: "Hay valores perdidos en la referencia o en la predicción de la clase". Comprobamos que, efectivamente, si sumamos todos los atributos de la matriz de confusión no obtenemos el total de atributos.

$$26399 + 4265 + 1351 + 4644 + 17456 + 514 + 2075 + 751 + 1451 = 58906 \neq 59400$$

¿A qué se debe esto? Ya sabemos que el conjunto de datos posee bastantes valores perdidos, estos no causarán problema a la hora de crear el modelo porque cuando nos encontremos con un valor perdido y tengamos que decidir cómo clasificarlo, lo haremos usando la última clase conocida. Es posible que alguna de las clases no aparezca en el conjunto de entrenamiento (functional needs repair, por estar menos representada) y, por tanto, cuando nos encontremos con un ejemplo de este tipo al realizar el test, nuestro modelo no pueda clasificarlo. ??? Esto no puede ser por esto, estamos creando las particiones manteniendo la distribución de clases.

Utilizaremos diferentes índices para poder interpretar la matriz de confusión, podemos verlos en la Tabla 4.

Tabla 4: CART - criterios de precisión

	TP	FP	TN	FN	TPR	TNR	PPV	Accur.	F1-score	G-mean
CART	17456	5016	31276	5158	0.77191	0.86179	0.77679	0.82728	0.77434	0.81561

En términos de precisión este algoritmo clasifica correctamente un 82,73 % de las instancias de la clase positiva.

La utilización de este algoritmo no precisó de ningún preprocesamiento, aceptó todos los atributos. Nos preocupamos porque no sabemos qué está pasando con las variables continuas, que los árboles de decisión no manejan bien, pero vemos en la descripción del nodo que los atributos numéricos (continuos) los dividió en dos subconjuntos a partir de su media para poder tratarlos de forma categórica.

Es un algoritmo robusto, como ya se comentó, es capaz de trabajar con valores perdidos.

Este modelo es fácilmente interpretable, dado un nuevo ejemplo podríamos partir del nodo raíz en el árbol obtenido y llegar a la clase con que se etiquetará siguiendo la rama del árbol correspondiente según el valor de cada atributo.

2.3. k-NN

A la hora de clasificar nuestra máxima es la semejanza, partiendo de este criterio el algoritmo más sencillo de entender podría ser el k-NN que realizará la predicción de una instancia en función a sus k vecinos más cercanos. El cálculo de la cercanía se hará en función a la distancia, por ello hay que discretizar y normalizar los datos. No podemos trabajar con variables categóricas, ya que no están ordenadas. Tampoco podemos trabajar con los datos sin normalizar porque daríamos más importancia a los atributos que tomaran mayores valores.

Comenzamos numerizando las variables. Para ello, utilizamos el nodo `Category to Number` que dada un atributo con n posibles categorías, asignará un número de 0 hasta $n-1$ a cada categoría. Por motivos de cómputo, excluiremos aquellas variables que exceden el máximo de categorías (`subvillage`, `wpt_name`). También excluimos la categoría "class", pues es la etiqueta que estamos tratando de asignar.

A continuación, normalizamos las variables mediante el nodo `Normalizer`. Ambas operaciones las realizamos antes de la validación cruzada, para el conjunto total de instancias, si normalizaramos solo en el conjunto de entrenamiento, podríamos encontrarnos con valores fuera del rango al realizar el test. Utilizamos el nodo `K Nearest Neighbor`, que contiene el algoritmo knn tomando como número de vecinos $k = 3$. Además, añadimos un `Column Rename` para que la columna con la predicción se llame "Prediction (class)", como en nuestros otros nodos, en vez de "Class [kNN]". El flujo en KNIME es el mostrado en la Figura 4.

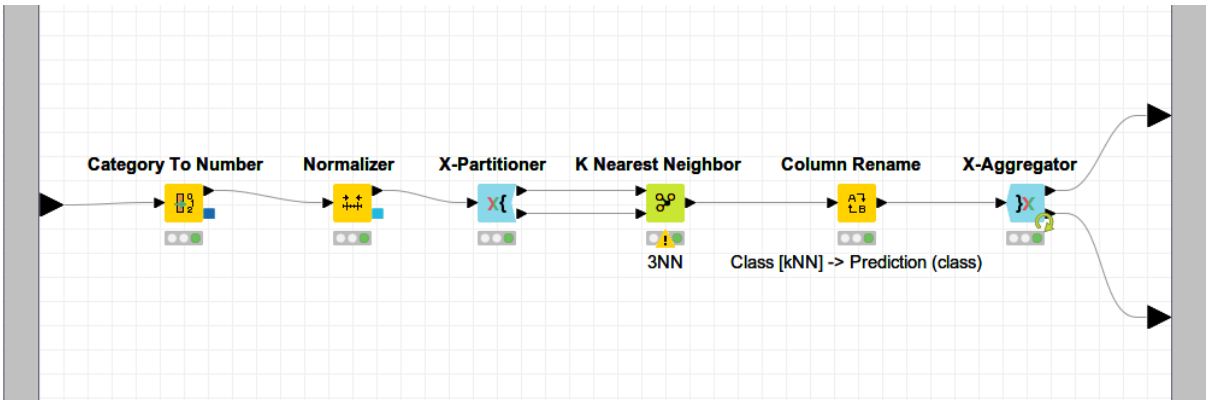

Figura 4: Metanodo 3-NN

La advertencia en el nodo `K Nearest Neighbor` indica que las filas con valores perdidos serán ignoradas. Este algoritmo no es robusto en el sentido de que no es capaz de manejar los valores perdidos. Este aspecto se trabajará en la Sección 5 y se compararán los resultados con los actuales.

A continuación, en la Tabla 5, se muestra la matriz de confusión para este algoritmo.

Tabla 5: 3-NN - Matriz de confusión

	functional	non functional	functional needs repair
functional	13863	1687	479
non functional	2454	7054	208
functional needs repair	1131	362	520

En la Tabla 6 se muestran los diferentes criterios de precisión para este algoritmo.

Tabla 6: 3-NN - criterios de precisión

	TP	FP	TN	FN	TPR	TNR	PPV	Accur.	F1-score	G-mean
3-NN	7118	1973	16069	2598	0.73261	0.89064	0.78297	0.83533	0.75695	0.80777

2.4. Red neuronal

Se ha escogido una red neuronal como siguiente algoritmo. Los algoritmos de redes neuronales se inspiran en las redes neuronales biológicas, tienen unos nodos (llamados neuronas) que se conectan con otros nodos, transmitiendo una señal que será un número real. Estas conexiones tienen un peso que se va ajustando en el proceso de aprendizaje.

En este caso también es necesario que los valores sean numéricos y estén normalizados. Además, aunque el nodo `Learner` permita ignorar los valores perdidos, si estos llegan al nodo `Predictor` obtendremos un error, luego es necesario manejar estos valores perdidos en nuestro preprocesamiento mínimo del algoritmo.

Comenzamos transformando a número los datos categóricos mediante el nodo `Category to Number`. Seguidamente, los normalizamos usando el nodo `Normalize` y mediante los nodos `X-Partitioner`, `X-Aggregator` realizaremos la validación cruzada. Tenemos que tratar los valores perdidos, para ello utilizamos el nodo `Missing Value` en la rama de entrenamiento y sustituimos los valores numéricos perdidos por su media y los categóricos por el más frecuente. En la rama de test usamos el nodo `Missing Value (Apply)` que aplicará las transformaciones del nodo `Missing Value`. Así, el metanodo en KNIME correspondiente al modelado de la red neuronal queda como se ve en la Figura 5.

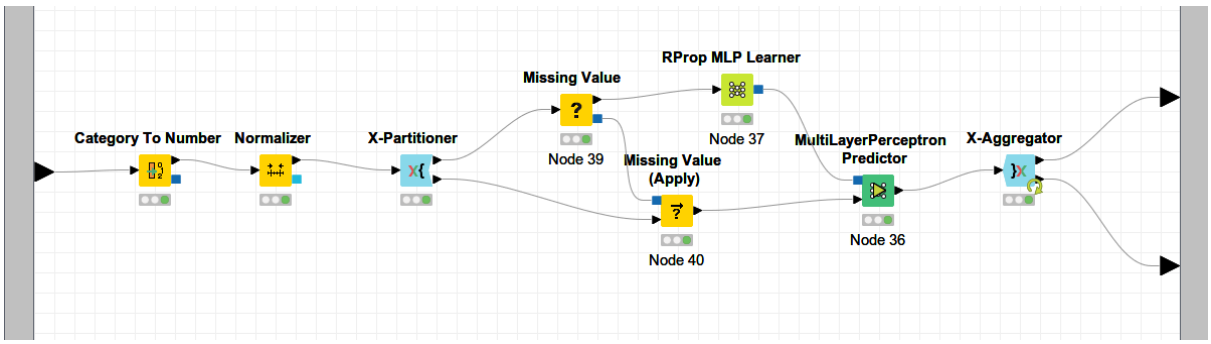

Figura 5: Metanodo MLP

Se utilizan los nodos `RProp MLP Learner` y `MultiLayerPerceptron Predictor` para modelar y probar este algoritmo. La red neuronal realizará un máximo de 100 iteraciones, tendrá una única capa oculta con 10 neuronas por capa.

La matriz de confusión obtenida la encontramos en la Tabla 7.

Tabla 7: MLP - Matriz de confusión

	functional	non functional	functional needs repair
functional	26964	5275	20
non functional	9040	13773	11
functional needs repair	3269	995	53

Las diferentes medidas de precisión asociadas a este algoritmo las vemos en la Tabla 8.

Tabla 8: MLP - criterios de precisión

	TP	FP	TN	FN	TPR	TNR	PPV	Accur.	F1-score	G-mean
MLP	13773	6270	30306	9051	0.60344	0.82858	0.68717	0.74207	0.64259	0.70711

2.5. Naive Bayes

El siguiente algoritmo elegido es uno basado en métodos bayesianos. Asume que los atributos son independientes y calcula la clase más probable condicionando el valor del resto de atributos.

Lo utilizamos en nuestra validación cruzada mediante los nodos `Naive Bayes Learner` y `Naive Bayes Predictor`, que dejamos con sus valores por defecto. El metanodo Naive Bayes creado en KNIME para este algoritmo se presenta en la Figura 6.

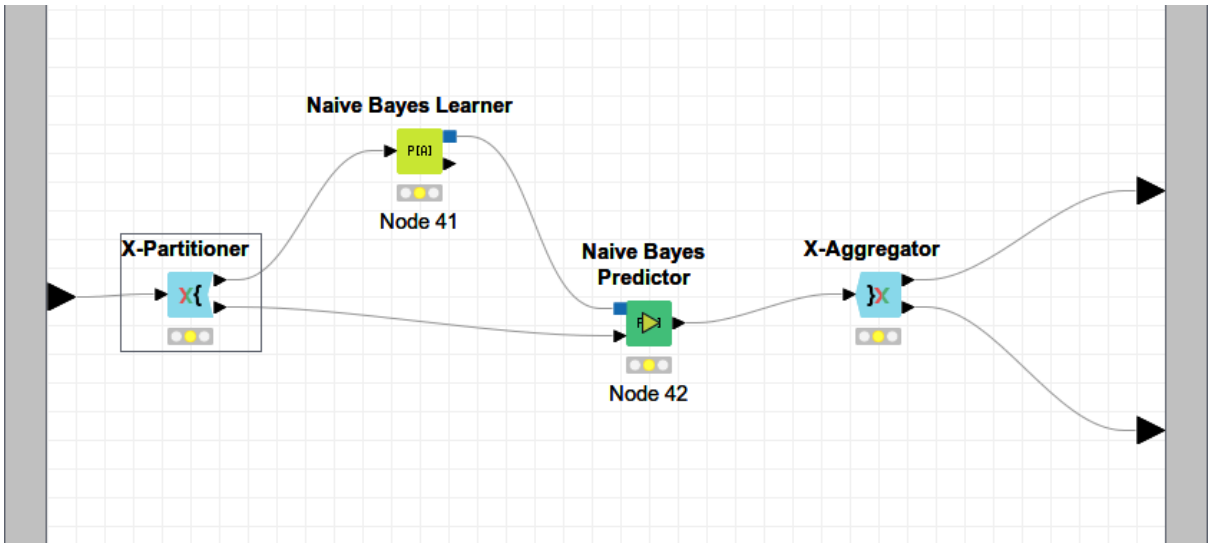

Figura 6: Metanodo Naive Bayes

Notamos que este algoritmo no necesita ningún preprocesamiento mínimo, trabaja tanto con variables numéricas como con variables categóricas, no es necesario que las normalicemos y aunque obtenemos una advertencia de que hay valores perdidos es capaz de obtener información sobre ellos que luego usará en el predictor.

La matriz de confusión obtenida mediante el uso de este algoritmo la podemos encontrar en la Tabla 9.

Tabla 9: Naive Bayes - Matriz de confusión

	functional	non functional	functional needs repair
functional	18426	7036	6797
non functional	3699	16087	3038
functional needs repair	648	800	2869

Las diferentes medidas de precisión aparecen en la Tabla 10.

Tabla 10: Naive Bayes - criterios de precisión

	TP	FP	TN	FN	TPR	TNR	PPV	Accur.	F1-score	G-mean
NB	16087	7836	28740	6737	0.70483	0.78576	0.67245	0.75466	0.68826	0.7442

2.6. Random Forest

Pasamos ahora a un multiclasificador, que combine varios clasificadores para tratar de mejorar la clasificación. Este primer multiclasificador será un ejemplo de *bagging*, cada clasificador se induce independientemente. Busca mejorar algoritmos inestables, que frente a pequeños cambios en el conjunto de entrenamiento puede provocar grandes cambios en la predicción.

Se ha escogido como ejemplo de *bagging* el algoritmo Random Forest, que realiza distintas clasificaciones con árboles más débildes (que no consideren todas las variables, sin poda) y con diferentes subconjuntos de los datos. Para realizar el modelo utilizaremos el nodo `Random Forest Learner`, que es probado mediante el nodo `Random Forest Predictor`.

El flujo en KNIME es el que vemos en la Figura 7 y los nodos han sido configurados para utilizar 100 clasificadores que utilicen el índice Gini para elegir los atributos por los que ramificar el árbol. Ha sido necesario usar el nodo `Domain Calculator`, sin restringir el número de posibles valores, para que tuviera en cuenta todas las variables.

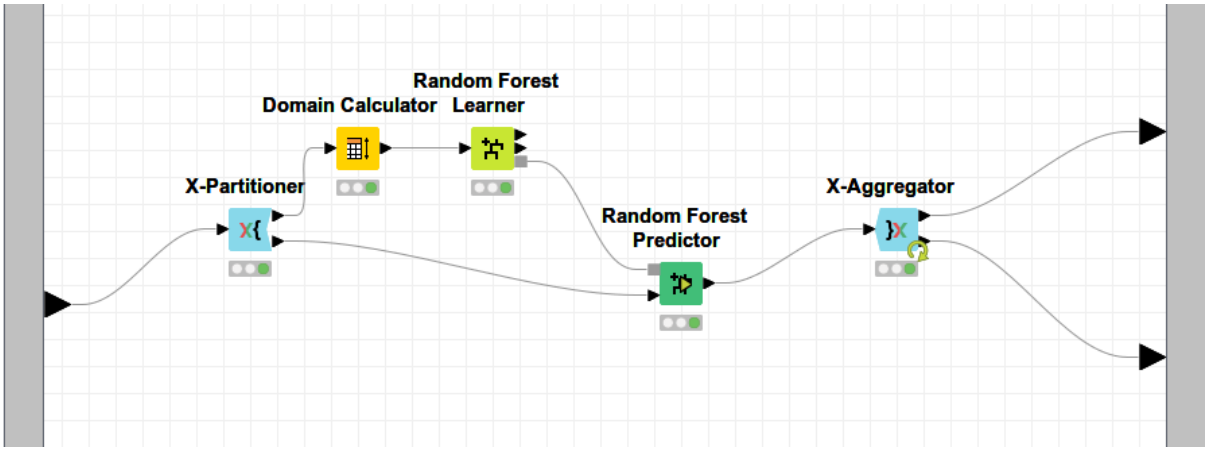

Figura 7: Metanodo Random Forest

En la Tabla 12 observamos la matriz de confusión obtenida mediante el uso de este algoritmo.

Tabla 11: -Matriz de confusión

	functional	non functional	functional needs repair
functional	29630	2126	503
non functional	5224	17341	259
functional needs repair	2484	548	1285

En la Tabla 12 se encuentran las medidas de precisión conseguidas por este algoritmo.

Tabla 12: Random Forest - Criterios de precisión

	TP	FP	TN	FN	TPR	TNR	PPV	Accur.	F1-score	G-mean
RF	17341	2674	33902	5483	0.75977	0.92689	0.8664	0.86268	0.80959	0.83918

Al ser un algoritmo que combina muchos árboles y estos admitir valores perdidos, Random Forest también admitirá valores perdidos. Sin embargo, la interpretabilidad de este modelo no es tan alta como lo era la de un único árbol de decisión. En el análisis compararemos este algoritmo con un único árbol de decisión.

2.7. Boosting

Probamos en este caso un algoritmo de *boosting*, esto es, un multiclasificador en el que cada clasificador tiene en cuenta los fallos del anterior.

Elegimos el nodo `XGBoosting Tree Ensemble Learner`, para realizar un modelo basado en árboles. Es necesario que numericemos las variables para no encontrarnos con problemas al predecir, así que añadimos el nodo `Category To Number` antes de realizar la partición. Vemos en la Figura 8 cómo quedó el flujo en KNIME necesario para ejecutar un algoritmo de *boosting*.

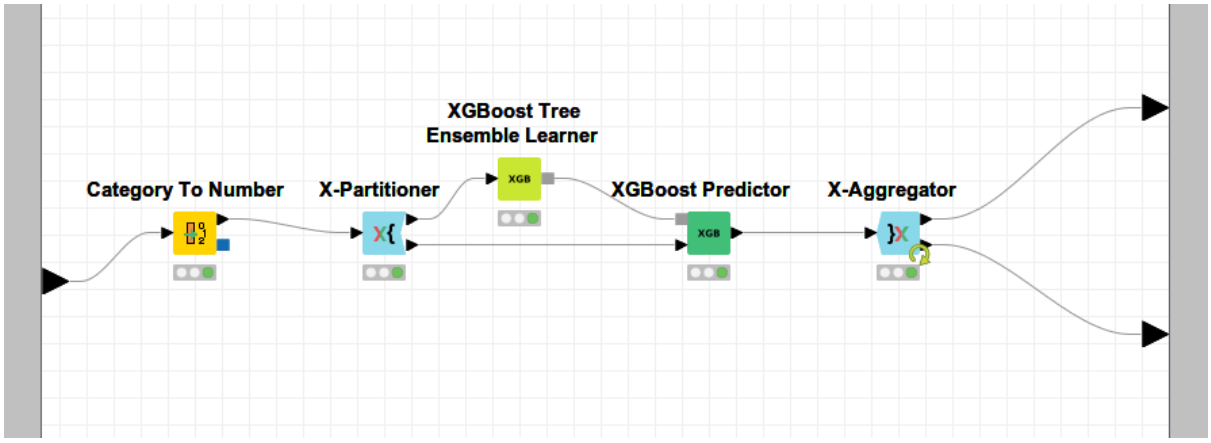

Figura 8: Metanodo XGBoost

Las tasas de clasificación obtenidas con este algoritmo las podemos visualizar en la Tabla 13.

Tabla 13: XGBoost - Matriz de confusión

row ID	functional	non functional	functional needs repair
functional	29290	2533	436
non functional	5438	17140	246
functional needs repair	2506	624	1187

Notamos que, aunque este algoritmo ha necesitado un mínimo preprocesado para admitir todas las variables del conjunto de datos, no fue necesario rellenar los valores perdidos. Comprobamos en 8 que este algoritmo soporta por defecto los valores peridos, durante el entrenamiento se trata con ellos.

Las medidas de precisión obtenidas para este algoritmo son las detalladas en la Tabla 14

Tabla 14: XGBoost - Criterios de precisión

row ID	TP	FP	TN	FN	TPR	TNR	PPV	Accuracy	F1-score	G-mean
XGB	17140	3157	33419	5684	0.75096	0.91369	0.84446	0.85116	0.79497	0.82834

3. Análisis de resultados

4. Configuración de algoritmos

5. Procesado de datos

6. Interpretación de resultados

7. Contenido adicional

8. Bibliografía