Prácticas Inteligencia de Negocio

Sergio Quijano Rey - 72103503k
sergioquijano@correo.ugr.es
5º Doble Grado Ingeniería Informática y Matemáticas
Grupo de prácticas 1

7 de noviembre de 2021

Índice

Índice de figuras

1. Introducción

En esta sección hablaremos de cada uno de los problemas abordados, tratando las particularidades de cada caso y algunas consideraciones en base a estas peculiaridades tratadas.

1.1. Información general

En todas las partes en las que necesitemos usar números aleatorios, usaremos la semilla 123456789 para poder reproducir bajo las mismas condiciones los experimentos y para que las comparaciones entre los distintos algoritmos sean lo más justas posible. En los siguientes subapartados, introduciremos los distintos problemas a tratar, sus características y problemas que puedan plantear.

En todos los *datasets* hemos usado la misma estructura jerárquica apoyándonos en los metanodos de `KNIME`. Esta estructura busca una mayor limpieza en el *"código"*. Dicha estructura se va a ir vislumbrando a lo largo de las secciones de estas memorias, donde incluiremos capturas de pantalla de los distintos *workflows* desarrollados.

1.2. *Heart Failure Prediction*

En primer lugar, la fuente original del *dataset* se puede encontrar en [1]. Aunque podemos estar trabajando con un *dataset* ligeramente modificado por los profesores de la asignatura, al igual que con el resto de *datasets* que estudiaremos en esta práctica.

En la propia página de la que se obtiene el dataset [1], se especifica que la tarea a resolver para este *dataset* es *"Create a model to assess the likelihood of a possible heart disease event. This can be used to help hospitals in assessing the severity of patients with cardiovascular diseases"*. Es decir, nuestra tarea es generar un modelo de clasificación para predecir, con los datos de entrada dados, si un paciente puede desarrollar algún problema de tipo cardiaco.

En el siguiente apartado, pasamos a comentar las particularidades de este *dataset*, información que hemos extraído con el análisis exploratorio hecho en `KNIME`:

1.2.1. Análisis Exploratorio de los Datos

Realizamos un pequeño *Análisis Exploratorio Inicial* (*EDA*) de los datos usando las herramientas que nos proporciona `KNIME`.

Como ya comentábamos en *"1.2.* Heart Failure Prediction*"*, buscamos predecir si una persona tendrá problemas de tipo cardiaco. Por tanto, la variable de salida con la que vamos a trabajar es `HeartDisease`. Consideraremos, por su mayor relevancia, como clase positiva, a la clase 1.

Lo primero que vemos es que tenemos 12 variables (11 variables de entrada más la variable de salida) y 918 filas (y por tanto, 918 ejemplos).

El *workflow* de mayor nivel, para este primer *dataset*, se muestra en la siguiente figura:

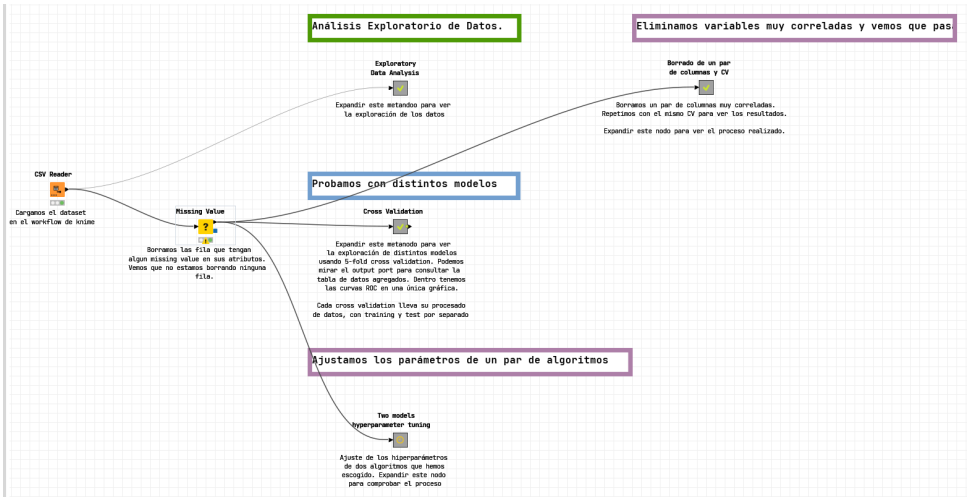


Figura 1: *Workflow* de más alto nivel para el primer *dataset*

La parte que ahora nos interesa es la de Análisis Exploratorio de Datos, que mostramos en la siguiente figura:

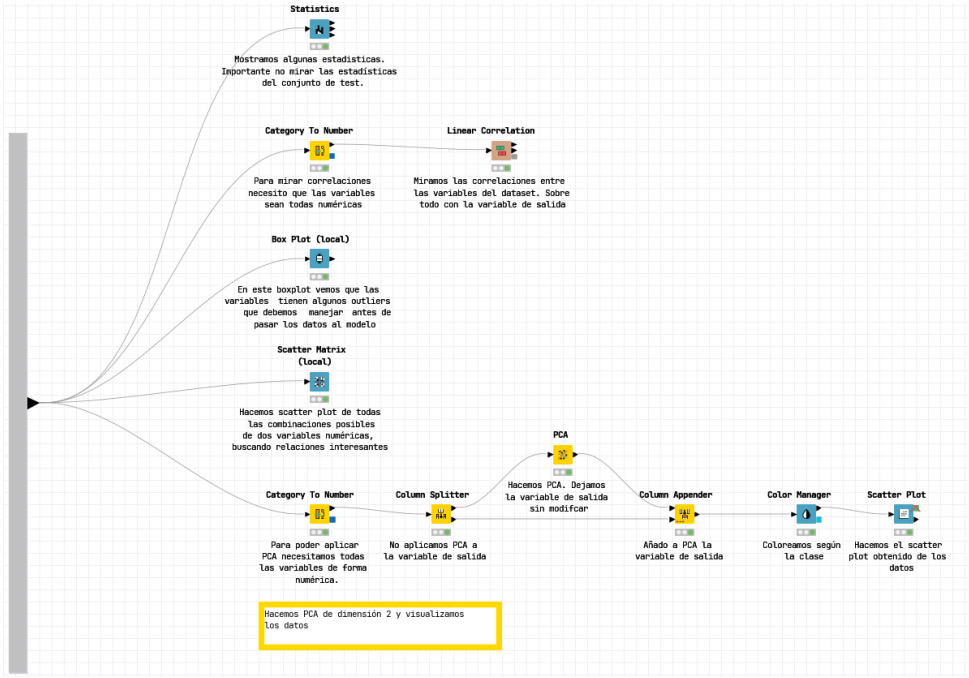


Figura 2: *Workflow* de Análisis Exploratorio de Datos para el primer *dataset*

Empezamos con el nodo de estadísticas, que nos muestra que tenemos 410 ejemplos para la clase de salida 0 y 508 para la clase 1. Por tanto tenemos un ligero desbalanceo (44.66 % para la clase 0 y 55.34 % para la clase 1), pero en este *dataset* no vamos a tratar dicho desbalanceo. En futuros *datasets* nos vamos a encontrar con clases mucho más desbalanceadas.

Dentro del anterior nodo vemos las distribuciones de las otras variables con las que trabajamos, sin llegar a conclusiones de gran relevancia, salvo razonamientos del tipo hay una característica que predomina en un valor sobre el otro valor en la población, o la población tiene una distribución de la variable normal o con cierta asimetría hacia un lado. Ejemplo de característica que predomina se muestra en la siguiente figura:

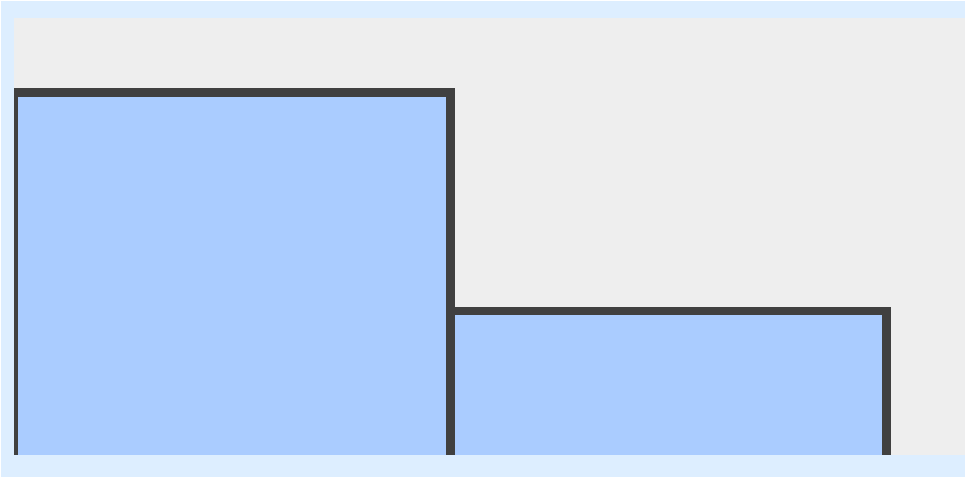


Figura 3: Distribución de la variable sexo. A izquierda, los hombres. A la derecha, las mujeres. Es claro que tenemos muchos más hombres que mujeres en nuestra población.

Esta información puede ser utilizada de forma experta en nuestros sistemas automáticos.

Sin embargo, por una cuestión de tiempo, no realizamos un ajuste tan a fondo de los modelos que vamos a generar (sobre todo teniendo en cuenta que practicar esto para los cuatro *datasets* es inviable. En otros *datasets* realizaremos una selección de variables usando otras técnicas).

Ejemplo de una distribución con cierta asimetría se muestra a continuación:

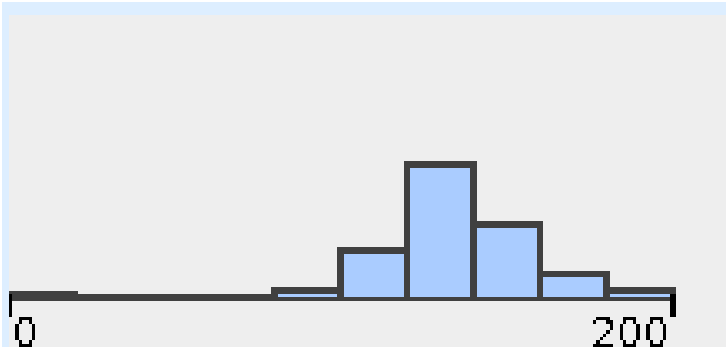


Figura 4: Distribución de la variable `RestingBP`. Podemos ver claramente cómo se acumula más hacia la derecha.

De nuevo, esta información es interesante para plantear los modelos de aprendizaje automático de una forma mucho más concienzuda, pero por la extensión de la práctica en técnicas y aspectos a tener en cuenta, no entramos en detalle en este aspecto.

Lo siguiente que hacemos en el Análisis Exploratorio de Datos es mostrar la matriz de correlaciones lineales, que presentamos a continuación:

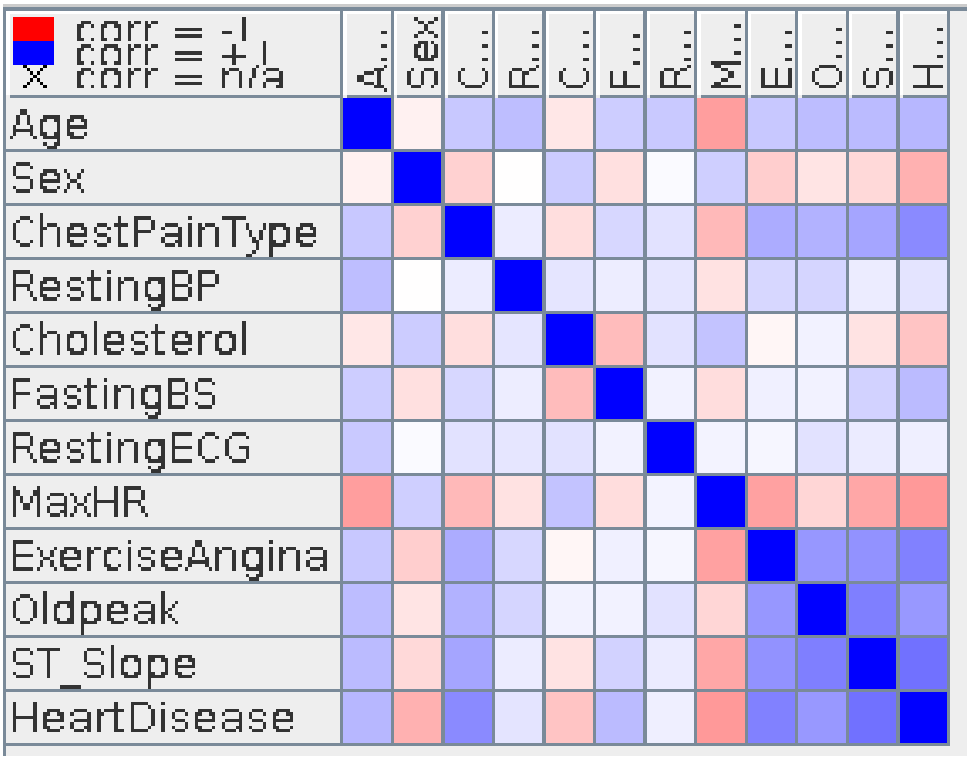


Figura 5: Matriz de correlaciones lineales. Un color azul significa correlación lineal positiva, y un color rojo significa correlación lineal negativa.

Vemos algunas variables correladas. Las más interesantes son el grupo que forman las variables `MaxHR` `ExerciseAngina`, `Oldpeak`, `ST_Slope` y la variable de salida `HeartDisease`. Sabiendo que estas variables están muy correladas, y que una de ellas es la variable de salida, podríamos probar a construir los modelos predictivos en base a este grupo de variables. Por tanto, estamos justificando el interés de que, más adelante, probemos a eliminar ciertas filas empleando esta matriz de correlaciones, y ver cómo afecta esto al rendimiento de nuestros modelos.

Mostramos ahora los *boxplots* de algunas variables, para ver que tenemos ciertos valores *outliers*:

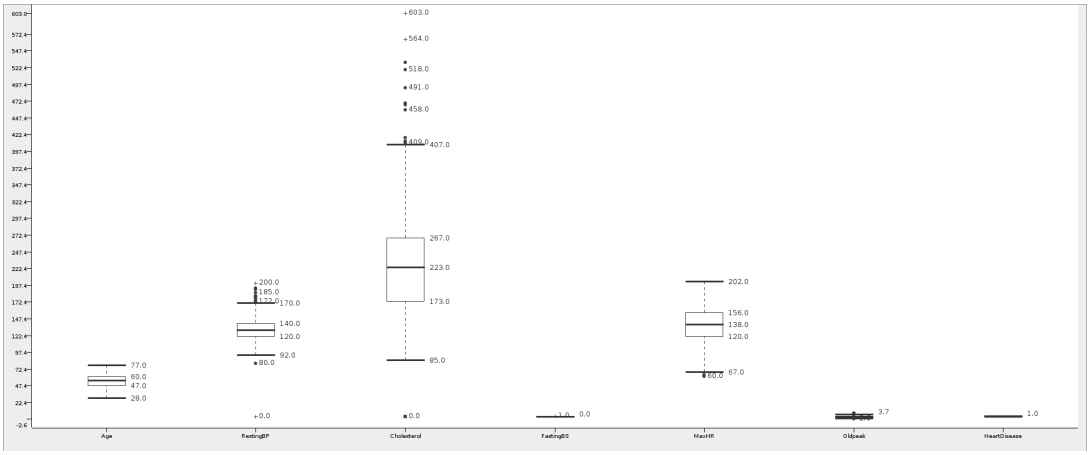


Figura 6: *Boxplots* de algunas de las variables con las que trabajamos. Queda claro que tenemos *outliers* (sabiendo que están alejados de la media más de 3 veces la desviación típica). Por tanto, tendremos que tratar de alguna forma estos *outliers*

También hacemos *plots* de todas las combinaciones posibles entre dos variables, obteniendo la siguiente gráfica:

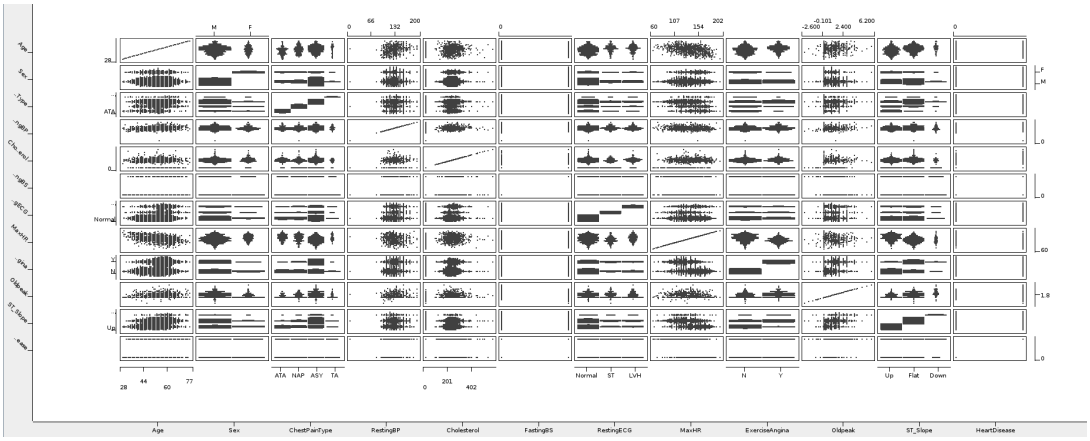


Figura 7: Gráficas de todas las posibles combinaciones de dos variables de nuestro dataset

De nuevo, como ya comentábamos para *"3. Distribución de la variable sexo. A izquierda, los hombres. A la derecha, las mujeres. Es claro que tenemos muchos más hombres que mujeres en nuestra población."*, podemos extraer de esta gráfica algunas conclusiones sobre la distribución de las variables y algunas relaciones que, por simpleza del problema y por falta de tiempo, no vamos a incluir en nuestro modelo.

En último lugar, probamos a aplicar *PCA* al *dataset* para obtener solo dos variables de entrada junto con la variable de salida, buscando sacar alguna información de alto nivel del problema. Esta técnica busca transformar el conjunto de datos con nuevas variables, de modo que estas variables no estén correlacionadas entre sí manteniendo el máximo posible de la varianza del conjunto de datos original. Este procedimiento se apoya en la descomposición en valores propios de la matriz de covarianzas [2]. Mostramos las dos variables obtenidas usando *PCA*, coloreando cada punto del plano $2D$ según a la clase a la que pertenecen:

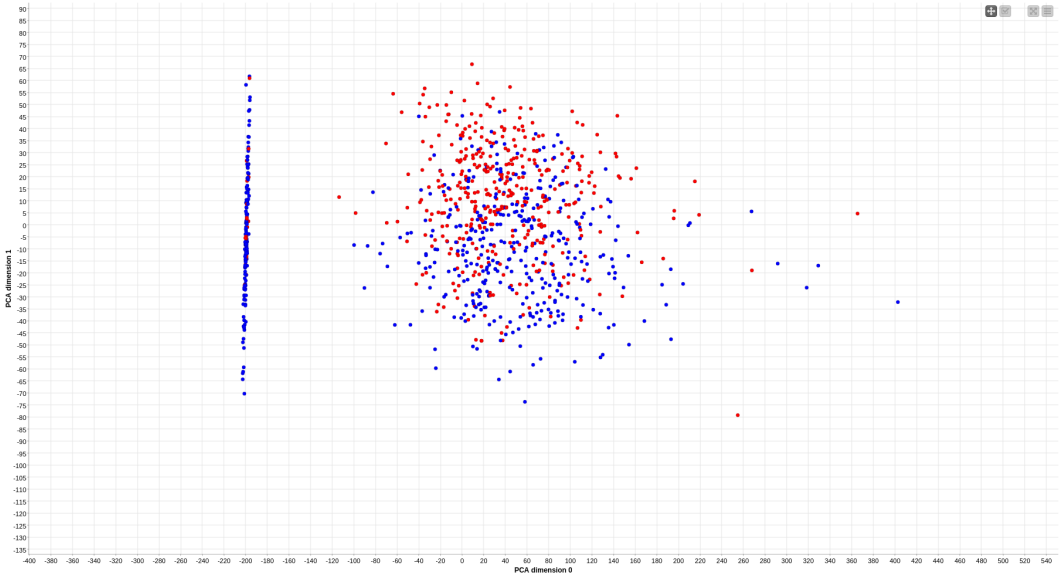


Figura 8: Resultado de aplicar *PCA* en dos dimensiones, coloreando cada punto según la clase a la que pertenecen

Los resultados obtenidos no nos invitan a usar *PCA* para usarlo como base a modelos de aprendizaje automático. Sin embargo es interesante que se ha obtenido una línea a la izquierda con clara predominancia de la clase representada por el color azul, mientras que a la derecha se obtiene un *clúster* ligeramente circular donde los datos de las dos clases están mezclados sin un patrón claro (quizás la clase roja tiene mayor predominancia en la parte superior). Estamos seguros de que aplicando *PCA* con dimensiones mayores obtendríamos un buen *performance* en los modelos de aprendizaje automático. Pero como para otros *datasets* es más interesante aplicar esta técnica, lo dejamos para más adelante.

En último lugar, tratamos los ***missing values***. En el *workflow* de más alto nivel, hacemos una primera aproximación al tratamiento de los *missing values*. Tratamos de borrar todas las filas que contengan *missing values*, pero no borramos ninguna fila. En un principio pensamos que no tenemos *missing values* en este *dataset*. Sin embargo, explorando la colección de datos comprobamos el siguiente hecho: la variable `Cholesterol` contiene *missing values* codificados con el valor 0. Para tratar estos *missing values*, en *Cross Validation* los marcaremos como tal, y los trataremos convenientemente en cada nodo de validación (para evitar hacer *data snooping*). Entraremos en detalles más adelante.

El nodo que usamos para realizar esta comprobación sobre los *missing values* se muestra en el *workflow* general que aparece en *"47.* decision tree *entrenado sobre todo el* dataset *tras aplicar el procesado anteriormente descrito"*.

1.3. Mobile Price Classification

En primer lugar, la fuente original del *dataset* se puede encontrar en [3]. De nuevo, podemos estar trabajando con un dataset ligeramente modificado por los profesores de la asignatura.

Según [3], la tarea a resolver es: *"In this problem you do not have to predict actual price but a price range indicating how high the price is"* para un cliente que quiere hacer la competencia a grandes compañías fabricantes de dispositivos móviles. Por lo tanto, tenemos que generar modelos que sean capaces de predecir dicho rango de precios.

En el siguiente apartado pasamos a comentar las particularidades de este *dataset*, particularidades que hemos extraído del *EDA* realizado sobre el conjunto de datos.

1.3.1. Análisis Exploratorio de los datos

Estamos usando el mismo *workflow* para el *EDA* que en el *dataset* anterior, así que no mostramos las capturas de los *workflows* de nuevo, sino que mostramos directamente los resultados, que es lo verdaderamente novedoso respecto al *dataset* anterior.

El primer problema que nos encontramos es que tenemos cuatro clases, en vez de dos clases. Por tanto, ya no estamos en un problema de clasificación binaria. Comentaremos más adelante, en *"2.3. Mobile Price Classification"*, cómo hemos trabajado este problema. Ahora mismo, en el *EDA*, este paso a clasificación multiclase no nos supone ningún problema.

La primera diferencia en los *workflows* con el *dataset* anterior es que estamos haciendo un *rename* de la clase positiva (que en nuestro caso consideramos la clase con valor 1, por los motivos que se explican en *"2.3. Mobile Price Classification"*), para que en *datasets* posteriores tengamos que realizar los mínimos cambios posibles en los distintos *worflows*. Este cambio de nombre se muestra enla siguiente figura:

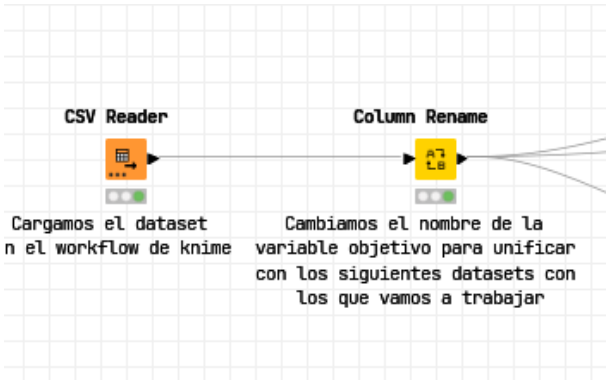


Figura 9: Cambiamos el nombre de la clase positiva al principio de todo nuestros procesos. Esto lo hacemos para que en los siguientes *datasets* tengamos que hacer los mínimos cambios posibles.

Lo primero que merece la pena comentar es que tenemos 20 variables de entrada, y una variable de salida. En nuestro *dataset* tenemos 2000 ejemplos, lo que ya supone un tamaño bastante más grande con respecto al *dataset* anterior.

En el nodo de estadísticas nos fijamos primeramente en la variable de salida. La siguiente

figura muestra que tenemos un balanceo perfecto entre las cuatro clases consideradas:

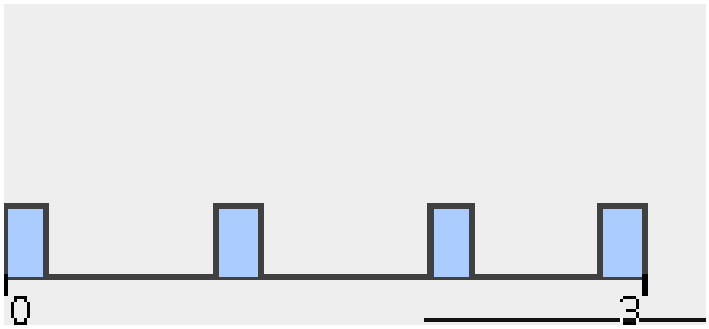


Figura 10: Las cuatro clases de salida están perfectamente balanceadas, con 500 ejemplos por cada clase

Para el resto de variables, podemos ver las distribuciones de los datos según cada variable, pero no sacamos conclusiones de utilidad. Las distribuciones son bastante normales, como se muestra en la siguiente figura:

Column	Min	Mean	Median	Max	Std. Dev.	Skewness	Kurtosis	No. Missing	No. +∞	No. -∞	Histogram
battery_power	501	1.238.5185	?	1.998	439.4182	0.0319	-1.2241	0	0	0	
blue	0.0	0.495	?	1	0.5001	0.02	-2.0016	0	0	0	
clock_speed	0.5	1.5223	?	3	0.816	0.1781	-1.3234	0	0	0	
dual_sim	0.0	0.5095	?	1	0.5	-0.038	-2.0008	0	0	0	
fc	0.0	4.3095	?	19	4.3414	1.0108	0.2771	0	0	0	
four_g	0.0	0.5215	?	1	0.4997	-0.0861	-1.9946	0	0	0	
int_memory	2	32.0465	?	64	18.1457	0.0579	-1.2161	0	0	0	
m_dep	0.1	0.5018	?	1	0.2884	0.0891	-1.2743	0	0	0	

Figura 11: Las distribuciones según cada variable son bastante normales. Tenemos distribuciones prácticamente uniformes, decrecientes, . . . No sabemos cómo podríamos explotar esta información.

Mostramos ahora la matriz de correlaciones lineales:

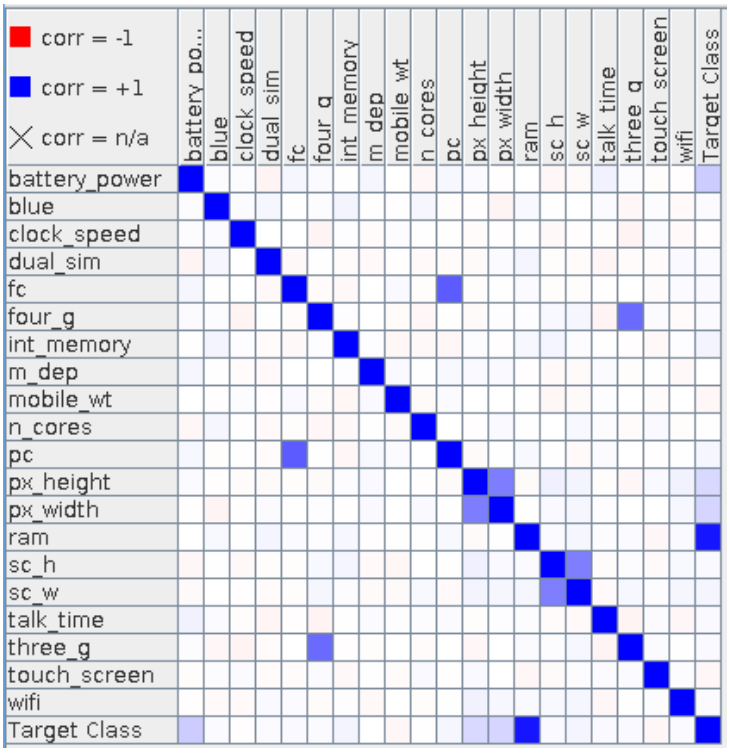


Figura 12: Matriz de correlaciones lineales. Un color azul significa correlación lineal positiva, y un color rojo significa correlación lineal negativa

Era de esperar la correlación entre la anchura y altura de píxeles (los móviles suelen guardar unos *aspect ratio* fijados), pero esto no nos es del todo útil, salvo para eliminar una columna, lo que no parece que vaya a tener un impacto enorme. También tenemos la misma correlación con las medidas en la unidad `sc_h, sc_w`, y de nuevo nos encontramos en la misma situación.

Lo que sí que es realmente interesante es que la variable que representa la memoria *RAM* correla mucho (0.917) con la variable de salida. Por tanto, podemos decir, adelantándonos a lo que comentaremos más adelante, que la memoria *RAM* es un factor decisivo para fijar el precio de un móvil.

Mostramos ahora los *boxplots* de algunas variables:

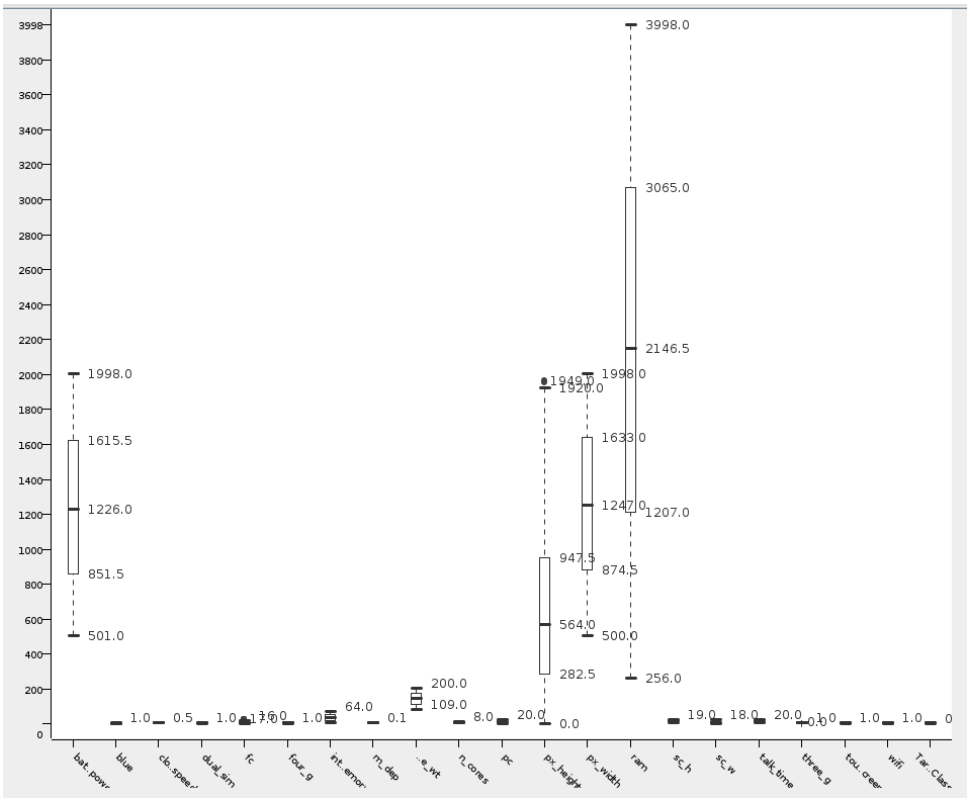


Figura 13: *Boxplots* de algunas variables

Queda claro que en este caso también tenemos bastantes *outliers* fijándonos solo en información unidimensional (variable a variable). Por tanto seguiremos borrando *outliers* usando el criterio $3 \cdot IQR$.

Mostramos ahora la gráfica de combinar algunas variables dos a dos, sin obtener resultados relevantes, como ya nos ha pasado previamente. Elegimos las variables que mostramos quitando aquellas que mostraban información del todo irrelevante, al menos según nuestra interpretación de esta gráfica:

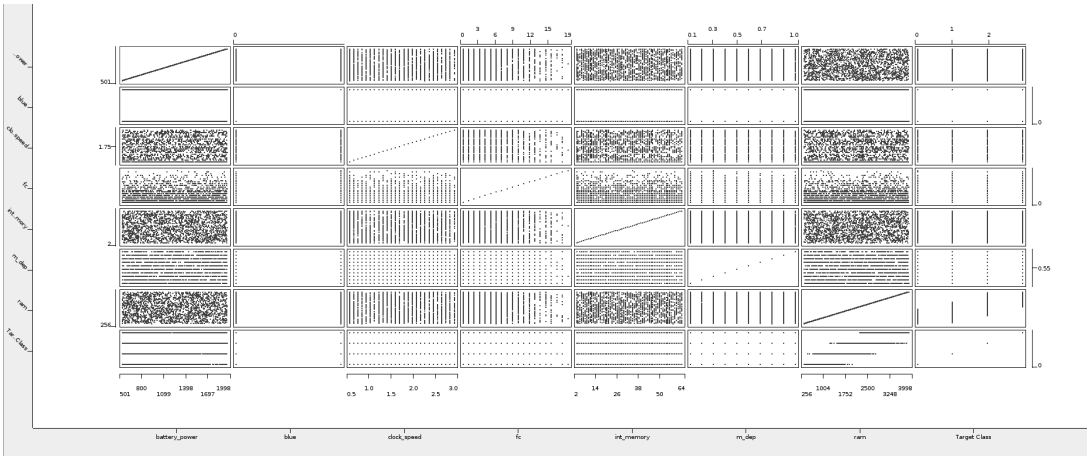


Figura 14: Gráfica de todas las posibles combinaciones de dos variables de nuestro dataset

En último lugar, aplicamos *PCA* para reducir nuestro conjunto de datos a tener dos variables

de entrada y la variable de salida. Al igual que hacíamos en el *dataset* anterior, mostramos los resultados visualmente, donde el color indica la clase a la que pertenece cada punto:

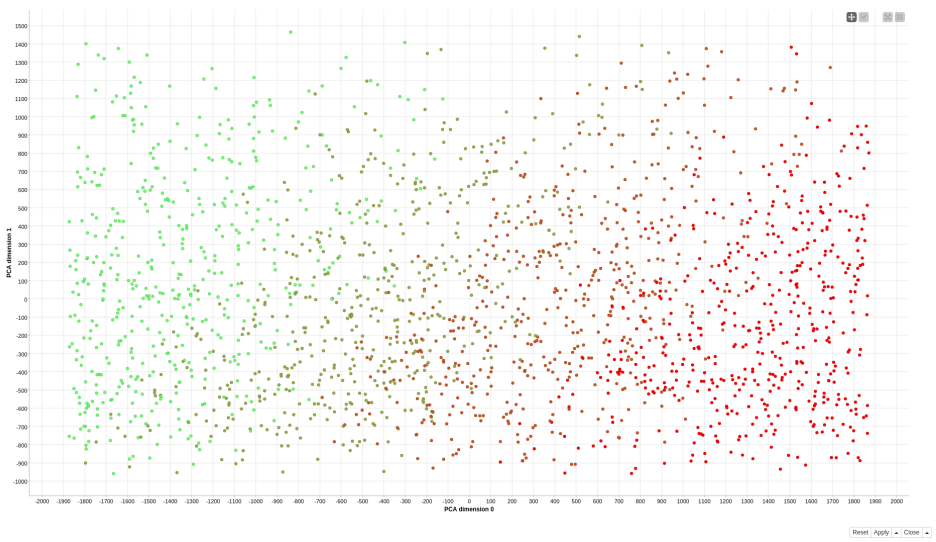


Figura 15: Resultado de aplicar *PCA* en dos dimensiones, coloreando cada punto según la clase a la que pertenecen

Para este *dataset* obtenemos un resultado muy interesante tras aplicar *PCA*. Gracias al coloreado vemos que los puntos tras transformarse se distribuyen de forma más o menos continua, en un gradiente que va en diagonal decreciente. Esto nos hace pensar que esta transformación, que pasa de 20 variables de entrada a únicamente 2, es útil para considerarse como paso previo a la generación de nuestros modelos de aprendizaje automático.

En último lugar, vemos que no tenemos *missing values* en el *dataset*. El nodo que borra las filas que contienen algún *missing value* deja el *dataset* con las mismas filas, y por tanto no detecta ningún *missing value*.

2. Resultados Obtenidos

2.1. Consideraciones iniciales

Para esta práctica, hemos considerado 7 modelos de aprendizaje automático distintos, que puedan ser interesantes para los cuatro problemas a los que nos vamos a enfrentar. Algunos de estos modelos pueden tener muy poco sentido para algún problema concreto. Sin embargo, elegimos los mismo 7 modelos para los 4 datasets, porque esto nos permitirá realizar una comparación entre ellos en distintos ambientes. De hecho, el que algunos modelos no tengan sentido en ciertas situaciones será una buena conclusión a extraer en el análisis posterior.

Los modelos considerados son:

1. Árbol de decisión construido con *AdaBoost*
2. Red Neuronal Simple
3. *Support Vector Machine*
4. *Support Vector Machine* con normalización
5. *K-NN*
6. *Naive Bayes*
7. *Random Forest*

Pensamos que con estos modelos tenemos una variedad suficiente para comparar en situaciones diferentes. Además, todos los los modelos tienen sentido para al menos uno de los *datasets* con los que vamos a trabajar.

En este proceso, usaremos un nodo al que llamamos *Custom Cross Validation*. Este nodo tiene la misma base que el nodo de `KNIME` *Cross Validation*. Sin embargo, añadimos dos nodos, uno para preprocesar el *fold* de entrenamiento y otro para aplicar dicho preprocesado al *test*, sin hacer *data snooping*. Es decir, las operaciones se calculan y aplican con *training*, y se aplican en *test* **usando el cálculo realizado sobre *training***.

En *"2.2. Heart Failure Prediction"* mostraremos de forma extensa los distintos *workflows* que componen nuestras siete validaciones cruzadas. Sin embargo, para los siguientes *datasets* no repetiremos el mismo proceso, para evitar ser demasiado repetitivos y extensos. Solo mostraremos aquellas partes en las que haya diferencias con lo mostrado en *"2.2. Heart Failure Prediction"* y con los *datasets* desarrollados previamente.

2.2. Heart Failure Prediction

2.2.1. *Workflows* empleados para *Cross Validation*

En primer lugar, tenemos un *metanodo* para encapsular todo el trabajo con los 7 algoritmos que hemos considerado. Este *metanodo* se mostraba en el *workflow* más general en *"47. decision tree entrenado sobre todo el* dataset *tras aplicar el procesado anteriormente descrito"*. Dentro de este nodo para *Cross Validation* tenemos la siguiente estructura:

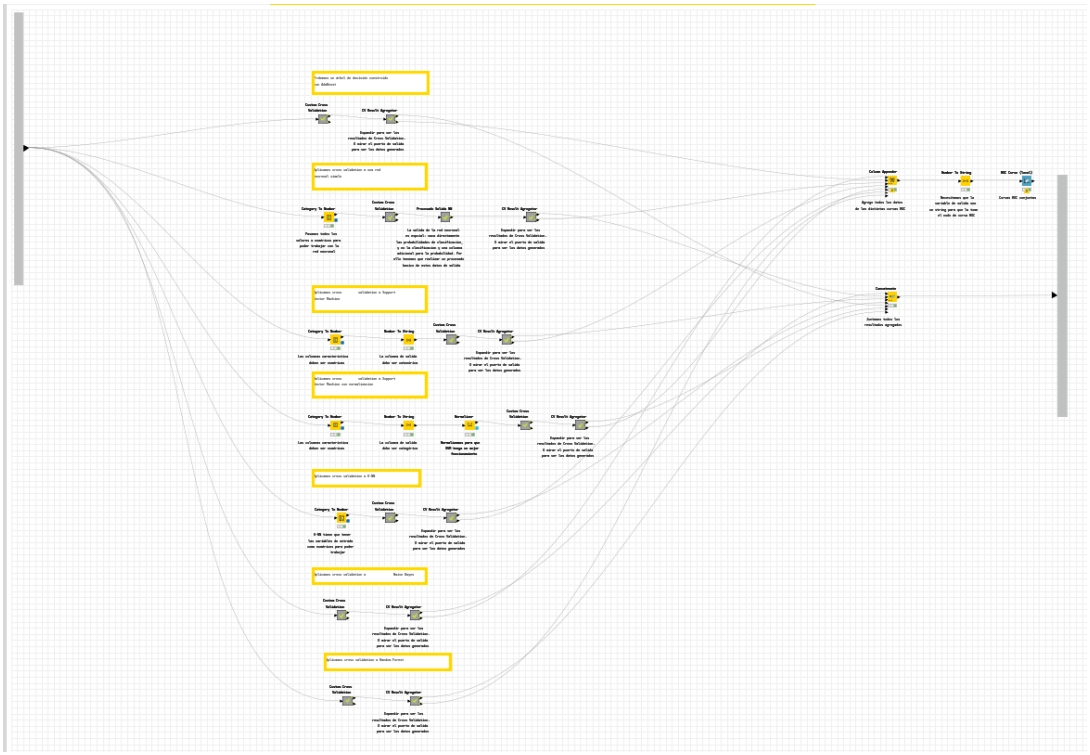


Figura 16: *Workflow* en el que realizamos todos los *Cross Validation* de los algoritmos considerados

Mostramos una captura con mayor *zoom* para que se vea claramente la estructura generada, dejando sin mostrar alguna de las filas correspondientes a algunos de los 7 modelos estudiados:

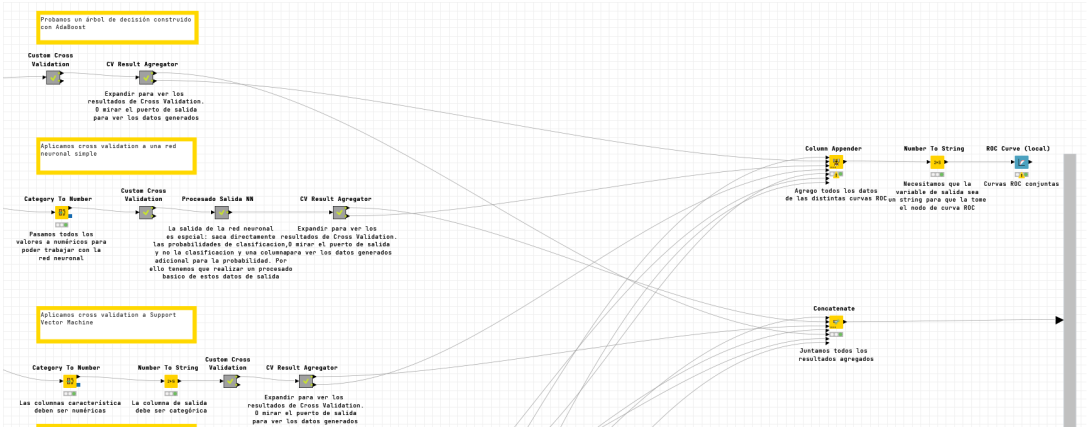


Figura 17: *Workflow* en el que realizamos todos los *Cross Validation* de los algoritmos considerados, con algo de *zoom* para que se aprecie mejor la estructura desarrollada

Mostramos el contenido del nodo *Custom Cross Validation* para *AdaBoost*. Este nodo *Custom Cross Validation* tiene ciertas diferencias como ya hemos mencionado en *"2.1. Consideraciones iniciales"*:

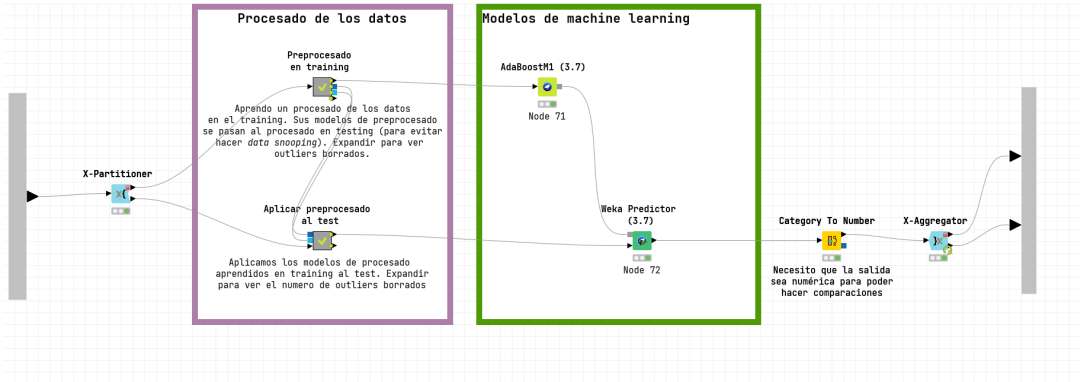


Figura 18: Nodo de *Cross Validation* para *AdaBoost*

Mostramos ahora el nodo para preprocesamiento en *training*:

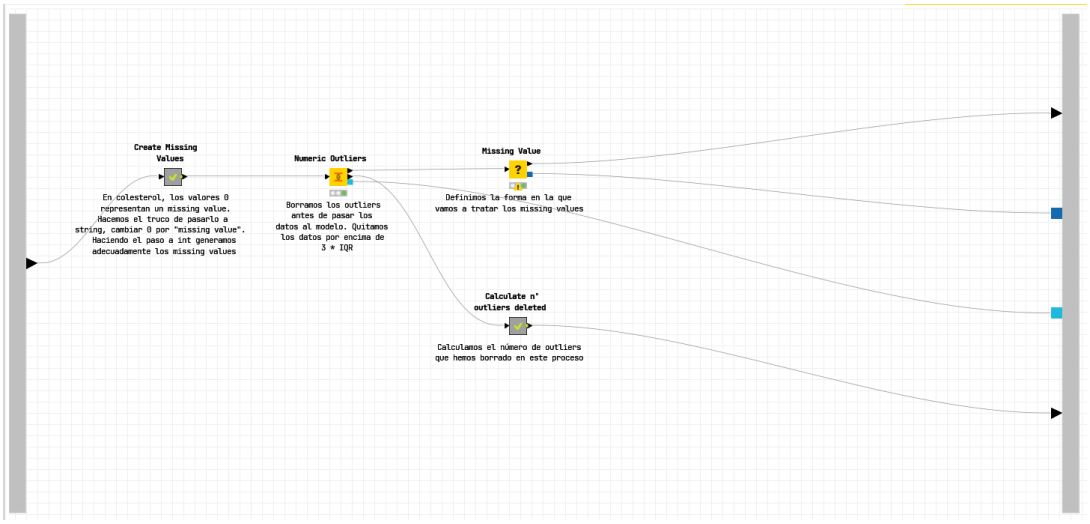


Figura 19: Preprocesamiento de los datos del *fold* de *training*

En esta figura se muestran dos partes fundamentales del procesamiento en este *dataset*. La primera es que, como ya comentábamos, tenemos *missing values* en la columna `Cholesterol` marcados con el valor 0. `KNIME` no detecta esto como *missing values*, así que empleamos el nodo *Create Missing Values* para marcarlos como tal.

En segundo lugar, tratamos los *outliers* para intentar reducir algo el ruido. Para ello, borramos aquellos valores que distan de la media más de tres veces la desviación típica (fundamentándonos en las propiedades de una distribución normal, en la que la mayoría de los datos están en el intervalo $[\mu - \sigma, \mu + \sigma]$ [4]).

Mostramos el nodo *Create Missing Values* para dejar claro el proceso que hemos descrito previamente:

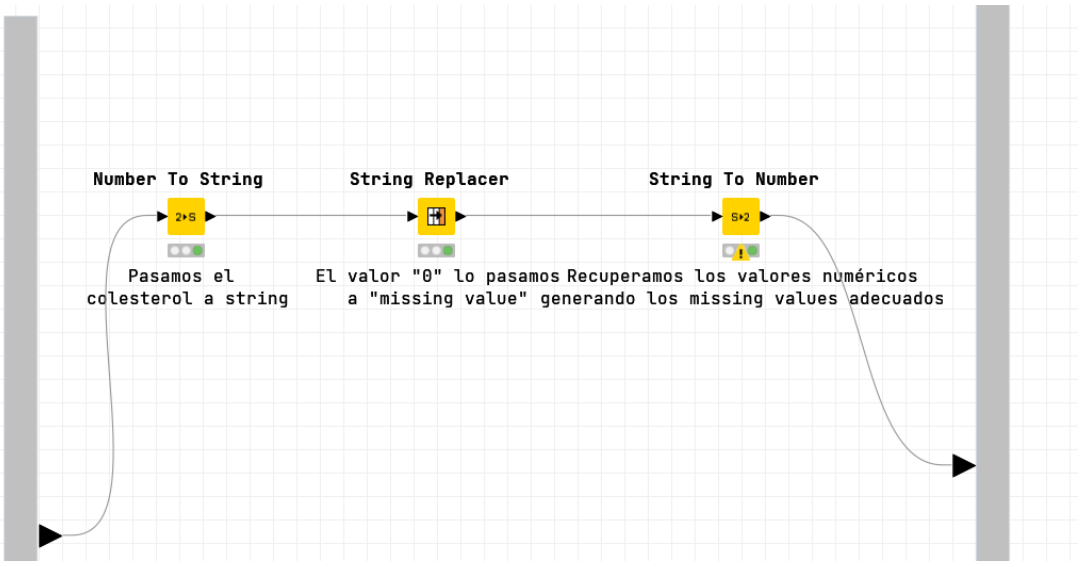


Figura 20: Nodo que usamos para crear los *Missing Values* que `KNIME` no detecta

Mostramos ahora el nodo para preprocesamiento en *test*:

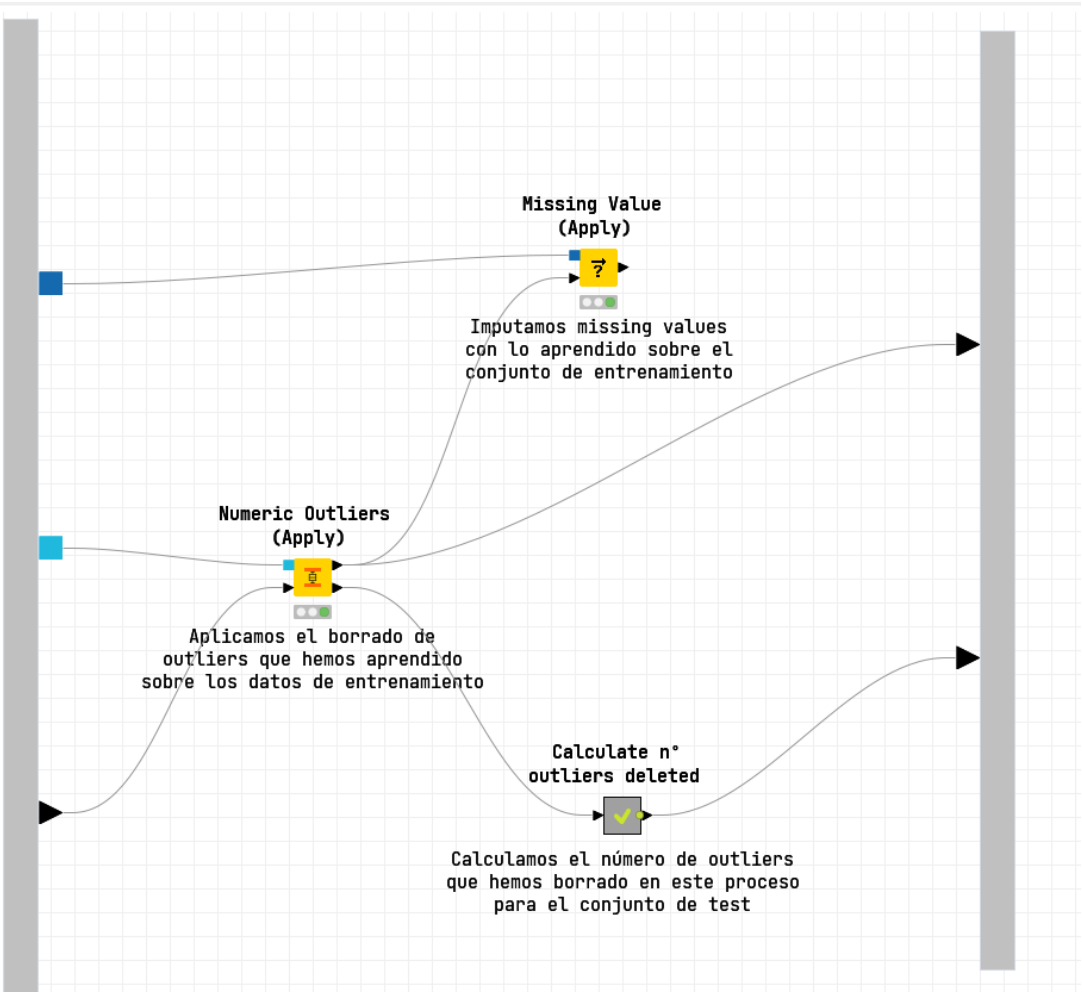


Figura 21: Preprocesamiento de los datos del *fold* de *test*

En *"21. Preprocesamiento de los datos del* fold *de* test*"* queda claro lo que ya comentábamos, preprocesamos usando los parámetros de procesamiento calculados sobre el *fold* de *training*. Esto es fundamental para **evitar hacer *data snooping***.

Además, de ahora en adelante, salvo que se diga lo contrario, estos dos nodos de preprocesamiento serán exactamente los mismos en todos los nodos de *Custom Cross Validation*, en todos los *dataset*, salvo que se indique lo contrario.

Como se muestra en *"17.* Workflow *en el que realizamos todos los* Cross Validation *de los algoritmos considerados, con algo de* zoom *para que se aprecie mejor la estructura desarrollada"*, estamos pasando la salida de *Custom Cross Validation* a un nodo de procesamiento, que mostramos a continuación:

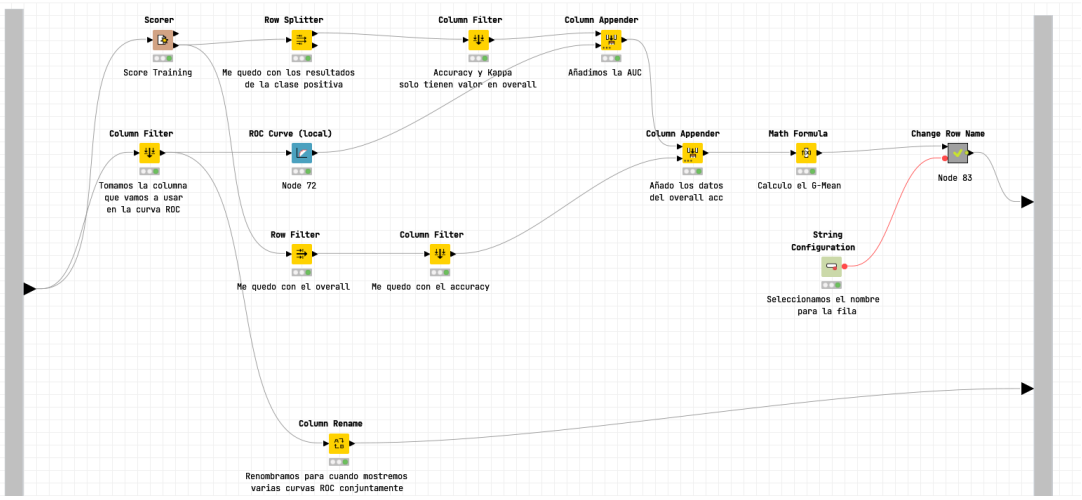


Figura 22: Nodo de procesamiento de la salida de *Custom Cross Validation*

En este nodo, tomamos la salida del *scorer* y la procesamos, tomando las métricas para la clase positiva y añadiendo el área debajo de la curva *ROC*. Además, calculamos la fórmula de la métrica *G-Mean*. También devolvemos las probabilidades en bruto para poder calcular las curvas *ROC* globales.

Se muestra también que estamos usando una `flow variable` para establecer el nombre del modelo cuyos resultados estamos procesando. Esto es útil para que en las tablas globales, cada fila esté etiquetada por el nombre del modelo que estamos tratando (esto es lo que consigue el nodo *Change Row Name*).

De nuevo, salvo que se diga lo contrario, todos los nodos de este tipo son idénticos.

Mostramos ahora el *workflow* para redes neuronales simples. Realizamos un preprocesado general básico antes de usar nuestro nodo *Custom Cross Validation* que mostramos en la siguiente figura:

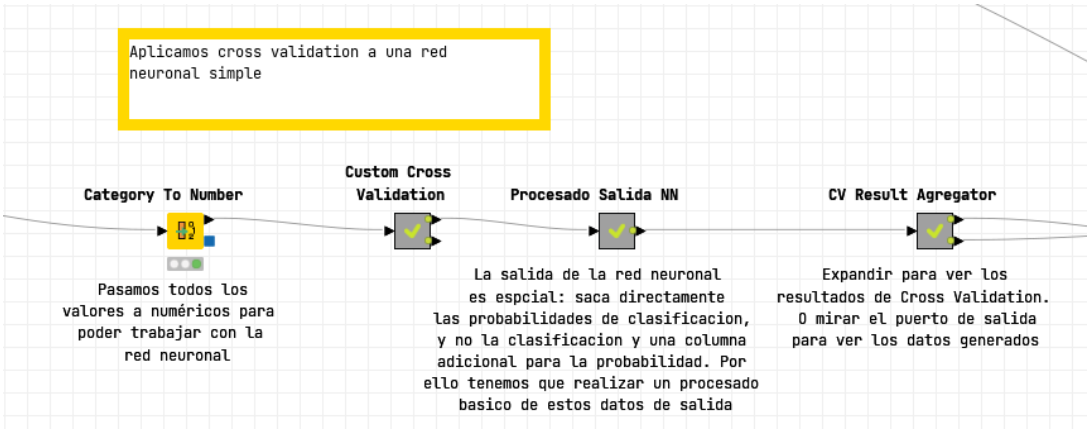


Figura 23: Preprocesado general previo a probar Redes Neuronales simples

Con esto queda claro que lo único que estamos realizando es pasar todas las variables categóricas a numéricas, pues esto es un requerimiento de las redes neuronales para que puedan funcionar. Mostramos el nodo de *Custom Cross Validation* para este caso:

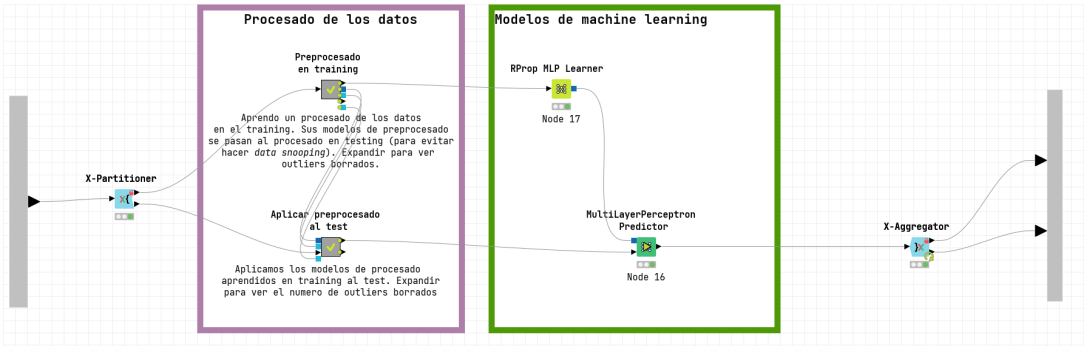


Figura 24: Nodo de *Cross Validation* para *Neural Net*

En este caso no estamos normalizando las variables, lo cual puede generar resultados no óptimos (pues las redes neuronales se benefician mucho de que las variables de entrada estén normalizadas. De hecho en *deep learning* se usa *batch normalization* para normalizar las salidas entre capas ocultas). Por simpleza del modelo no hacemos esta normalización para el *dataset*, pero en siguientes *datasets* sí que haremos la normalización.

La salida de las redes neuronales suponen un caso concreto que tratar. La red neuronal no nos da la clasificación directamente, sino un valor $x \in [0, 1]$. Tenemos que tratar la salida a mano. Para ello duplicamos la salida de este valor x que representa la probabilidad de clasificación que usaremos para las curvas *ROC*. En una de las dos ramas, colapsamos ese valor al conjunto $\{0, 1\}$ redondeando, y consideramos eso la salida de clasificación. Mostramos ese proceso, que se hace en el nodo de *Procesado Salida NN*, en la siguiente figura:

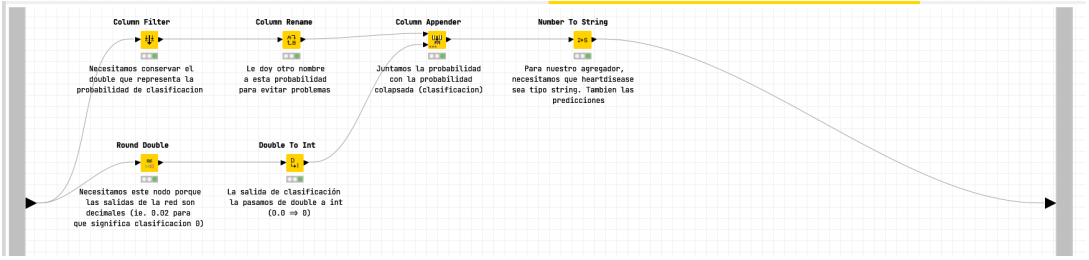


Figura 25: Procesado de la salida de *Neural Net*

Con esto, usamos el nodo que procesa la salida de *Cross Validation* que ya hemos mostrado previamente, y que no volvemos a mostrar para evitar ser repetitivos.

Mostramos ahora el preprocesado general para *Support Vector Machine*. Consiste en pasar a variables numéricas todas las variables menos la de salida, que debe ser de tipo categórica:

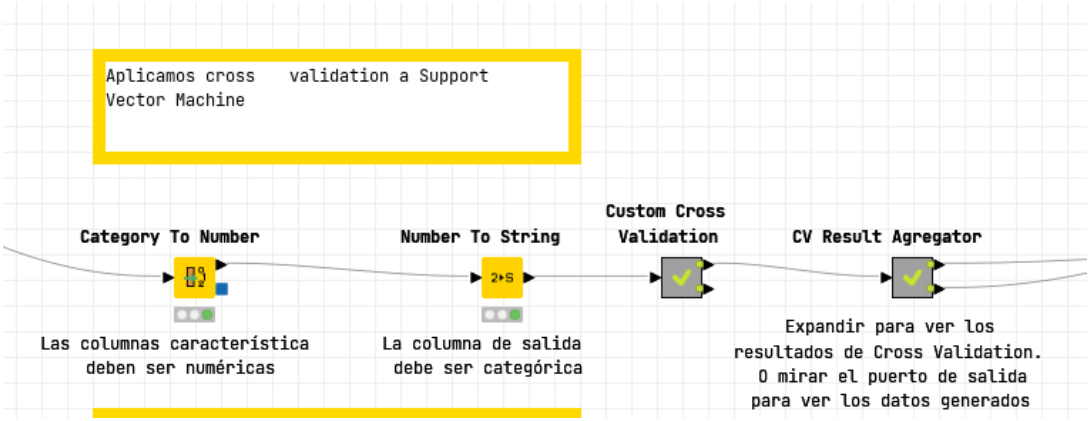


Figura 26: Preprocesado general previo a probar *Support Vector Machine*

Mostramos ahora el nodo *Custom Cross Validation* para este modelo:

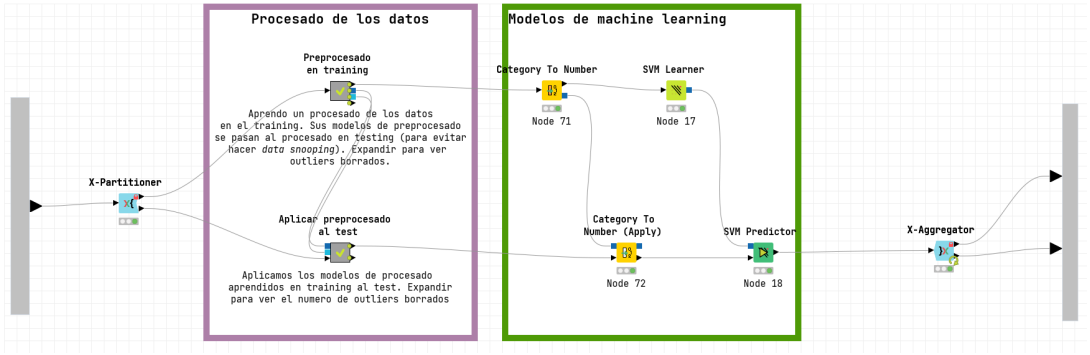


Figura 27: Nodo de *Cross Validation* para *Support Vector Machine*

En este caso, nos aseguramos de nuevo que estamos usando variables numéricas para las variables de entrada. En este *dataset*, usamos los nodos de KNIME para *SVM*. En otros *datasets* donde la eficiencia sea clave, pasaremos a usar los nodos de *Weka* que aparentemente funcionan de forma más rápida.

En el caso de *Supoort Vector Machine* con normalización, lo único que cambiamos es el preprocesado general que aplicamos antes de *Custom Cross Validation*. Mostramos esto a continuación:

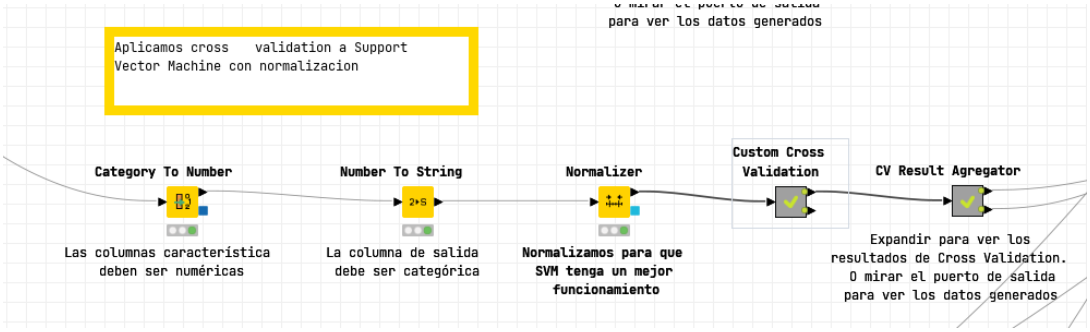


Figura 28: Preprocesado general previo a probar *Support Vector Machine* con normalización. Esto es lo único que cambia respecto al modelo anterior: *SVM* sin normalización

Mostramos ahora el preprocesado general para *K-NN*. *K-NN* tiene que trabajar con todas las variables de entrada en formato numérico, y esto es lo que hacemos:

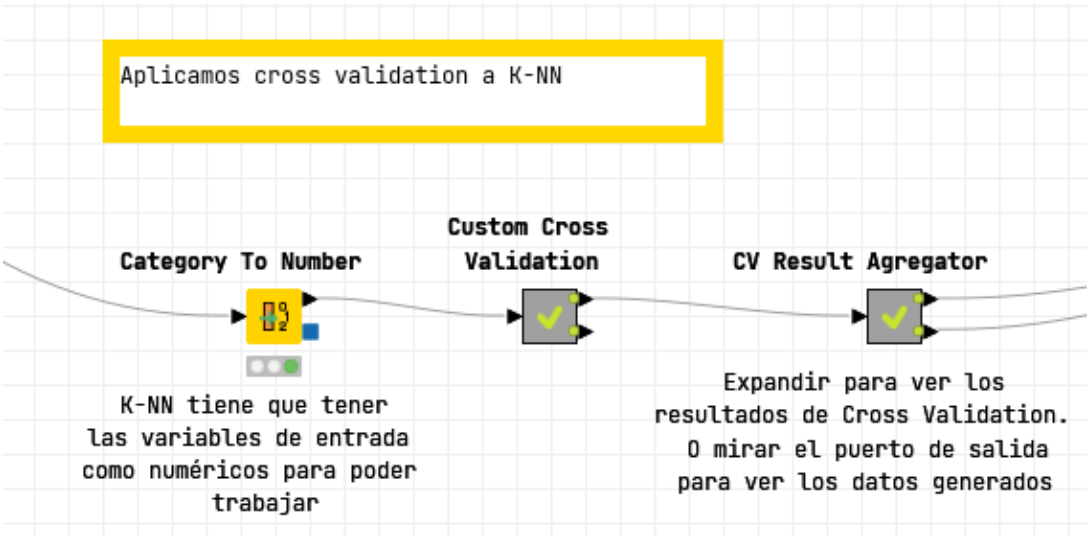


Figura 29: Preprocesado general previo a probar *K-NN*

Dentro del nodo de *Custom Cross Validation* lo que hacemos es, de nuevo asegurar que tenemos las variables en el formato correcto, y formatear la salida de *K-NN* para que el nodo *X-Aggregator* pueda funcionar correctamente. Mostramos esto a continuación:

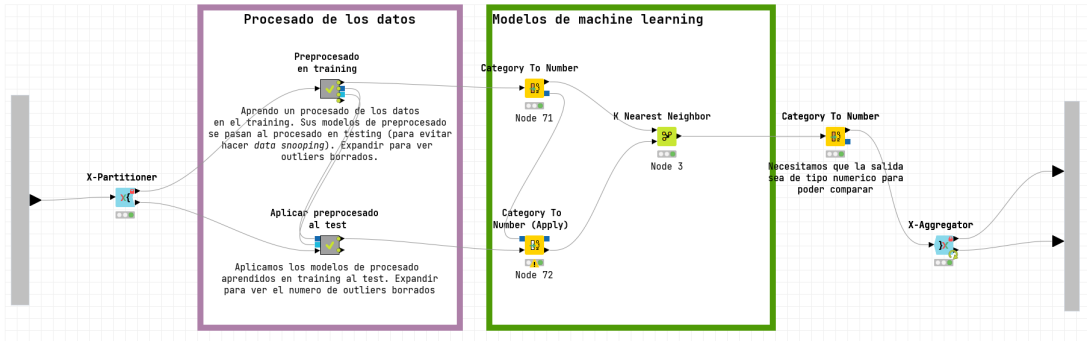


Figura 30: Nodo de *Cross Validation* para *K-NN*

Mostramos ahora el preprocesado general para *Naive Bayes*. En este caso, no estamos realizando ningún preprocesado general:

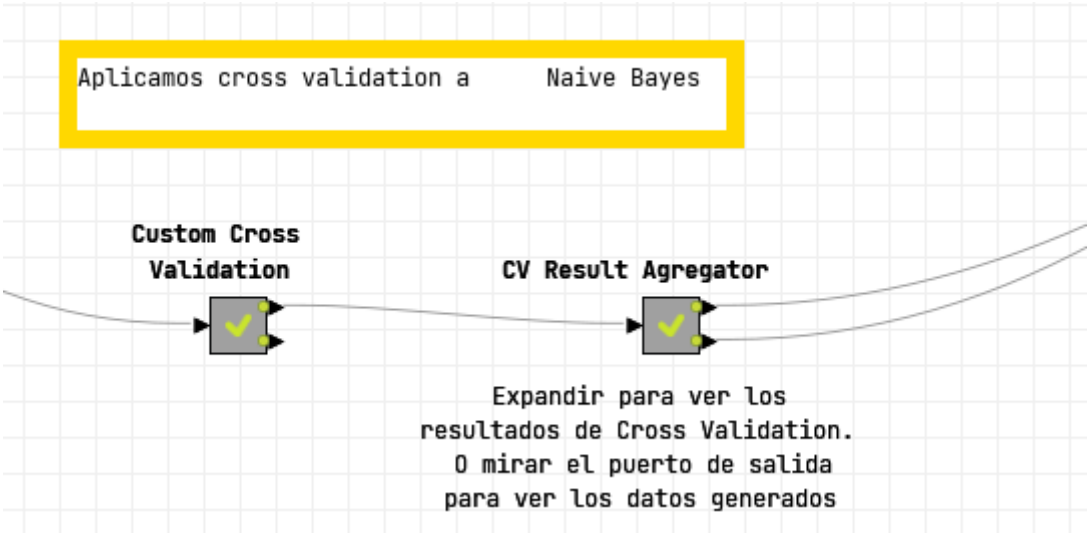


Figura 31: Preprocesado general previo a probar *Naive Bayes*. En este caso no estamos realizando ningún tipo de preprocesado previo a *Custom Cross Validation*

Mostramos lo que hacemos dentro del nodo *Custom Cross Validation*. No realizamos ningún preprocesado adicional además de los que ya hemos comentado previamente. La salida la pasamos a valores numéricos, para que el nodo `X-Aggregator` pueda trabajar correctamente:

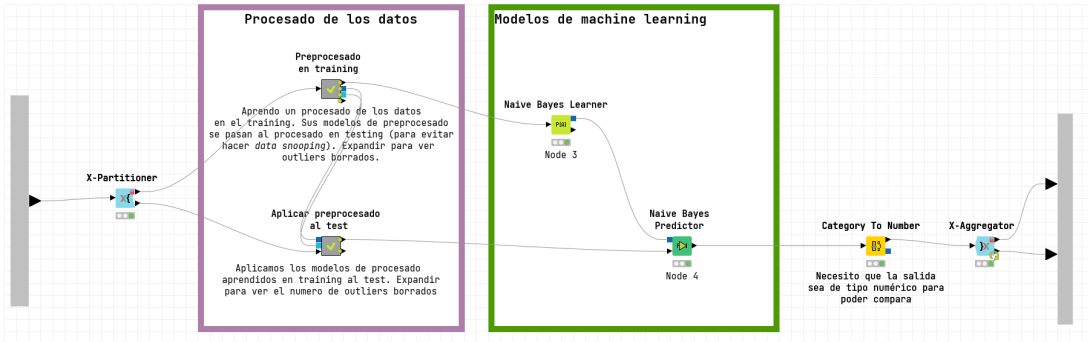


Figura 32: Nodo de *Cross Validation* para *Naive Bayes*

Terminamos esta sección mostrando el proceso para Random Forest. De nuevo, no realizamos un preprocesado previo general para aplicar el algoritmo, como se muestra a continuación:

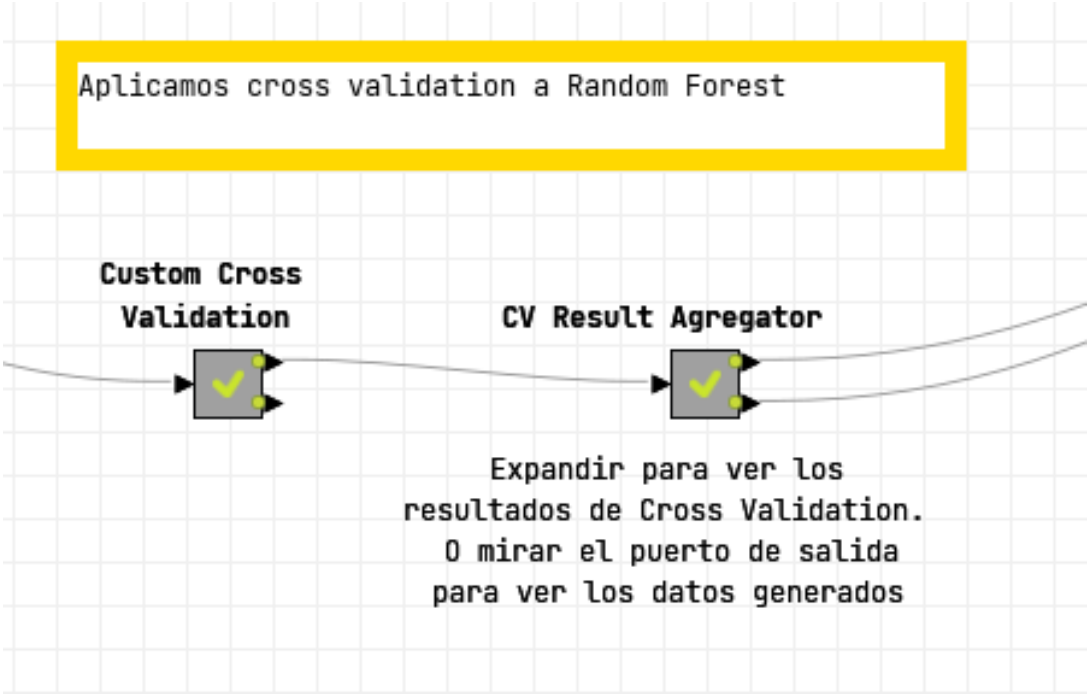


Figura 33: Preprocesado general previo a probar *Random Forest*. En este caso tampoco estamos realizando ningún tipo de preprocesado previo a *Custom Cross Validation*

Y de nuevo, nos encontramos con la misma situación dentro de *Custom Cross Validation*. Simplemente pasamos a variable numérica la salida para que funcione bien el nodo de `X-Aggregator`. Mostramos esto en la siguiente figura:

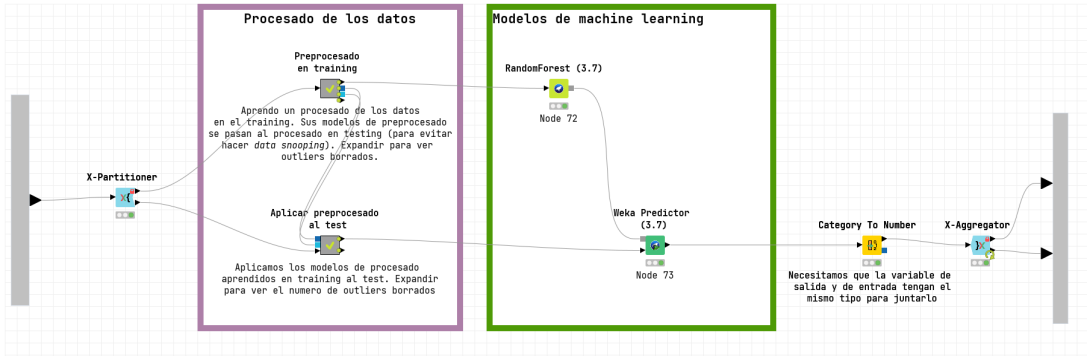


Figura 34: Nodo de *Cross Validation* para *Random Forest*

2.2.2. Mostrando los resultados obtenidos

En primer lugar, mostramos la tabla que resume todos los resultados obtenidos tras el proceso de *Cross Validation*:

Algo Name	TP	FP	TN	FN	Recall	Precision	Sensitivity	Specificity	F-measure	AUC	Accuracy	G-Mean
Naive Bayes	212	106	247	79	0.73	0.67	0.73	0.70	0.70	0.92	0.71	0.71
Neural Net	212	42	179	48	0.82	0.83	0.82	0.81	0.82	0.88	0.81	0.81
Support Vector Machine	6	1	352	285	0.02	0.86	0.02	1.00	0.04	1	0.56	0.14
Support Vector Machine + Normalization	141	33	332	195	0.42	0.81	0.42	0.91	0.55	1	0.67	0.62
K-NN	155	161	192	136	0.53	0.49	0.53	0.54	0.51	0.72	0.54	0.54
AdaBoost	208	102	251	83	0.71	0.67	0.71	0.71	0.69	0.92	0.71	0.71
Random Forest	215	107	246	76	0.74	0.67	0.74	0.70	0.70	0.92	0.72	0.72

Cuadro 1: Resultados de *Cross Validation*, para los distintos algoritmos estudiados, en el primer *dataset*

Mostramos ahora una tabla con las medidas de complejidad de cada uno de los algoritmos estudiados:

Algo Name	Medida de Complejidad
Naive Bayes	20 valores únicos por atributo
Neural Net	1 hidden layer, 10 neuronas
Support Vector Machine	Kernel RBF $\sigma = 2.0$
Support Vector Machine + Normalization	Kernel RBF $\sigma = 2.0$
K-NN	10 vecinos
AdaBoost	10 iteraciones
Random Forest	50 árboles, profundidad no acotada

Cuadro 2: Medidas de complejidad de los algoritmos considerados

2.2.3. Mostrando las curvas ROC obtenidas

En todos los algoritmos, hemos tomado las probabilidades de clasificación, con las que hemos mostrado las curvas ROC para la clase positiva. Estas curvas las mostramos tanto de forma individual como de forma conjunta. Empezamos mostrando la gráfica con todas las curvas ROC agrupadas:

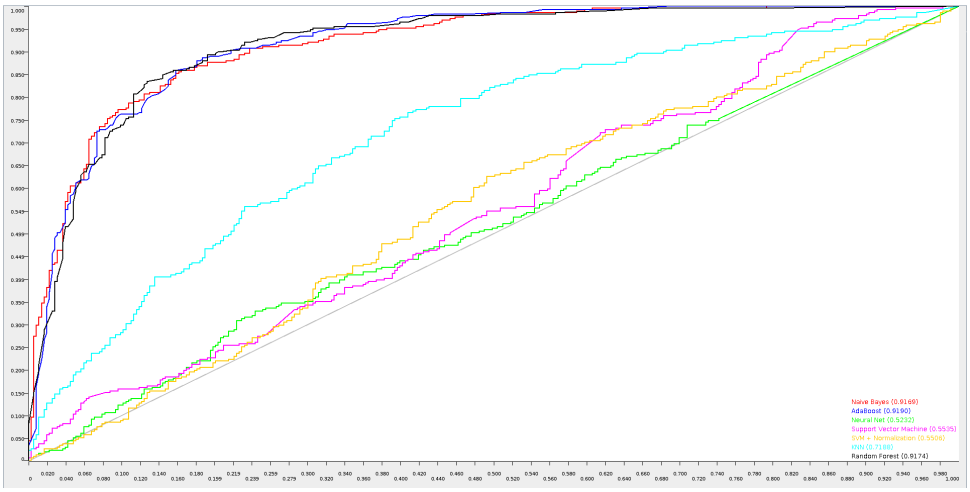
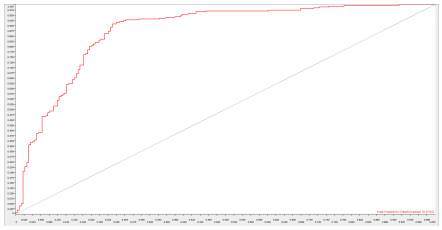
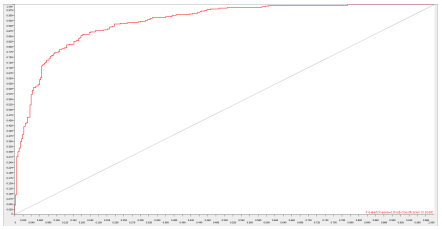


Figura 35: Todas las curvas ROC de los algoritmos estudiados

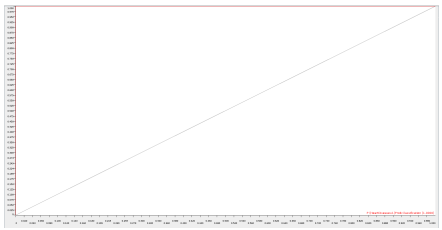
Algunos algoritmos se muestran de forma correcta. Es el caso de Naive Bayes, Random Forest, AdaBoost y K-NN. Los otros tres algoritmos (SVM en sus dos variantes y la red neuronal) se muestran de forma incorrecta. Esto se explica porque estos algoritmos no tratan de la misma forma los *missing values*, en muchos casos ignorándolos. Por tanto, obtenemos dos tipos de tablas que juntar, un tipo con todas las filas originales, y otro tipo de tabla en la que faltan algunas filas. Por tanto la visualización no es correcta. Es por esto que mostramos a continuación todas las curvas de forma individual:



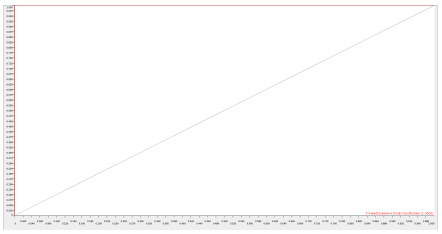
(a) Curva ROC para Redes Neuronales



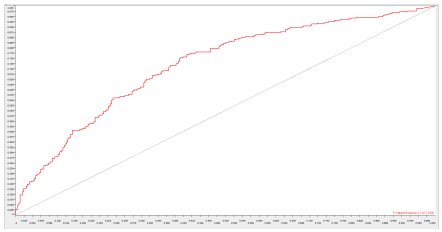
(b) Curva ROC para Naive Bayes



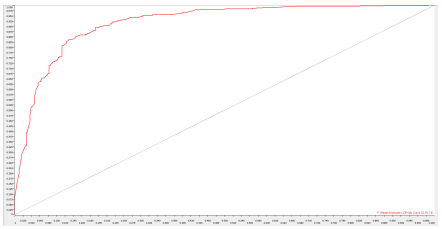
(c) Curva ROC para SVM



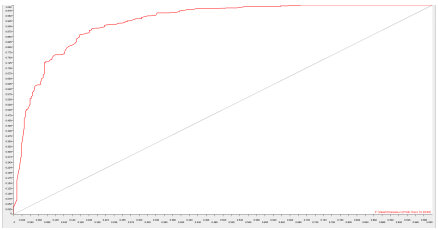
(d) Curva ROC para SVM con normalización



(e) Curva ROC para K-NN



(f) Curva ROC para Random Fores



(g) Curva ROC para Adaboost

Figura 36: Todas las curvas ROC para el primer *dataset*, mostradas de forma individual

Con esta visualización sí que se pueden extraer más conclusiones sobre el comportamiento de los algoritmos, como desarrollaremos más adelante.

2.3. Mobile Price Classification

2.3.1. *Workflows* empleados para *Cross Validation*

Como ya hemos comentado previamente, no vamos a mostrar capturas de pantalla de los *workflows* de cada *Custom Cross Validation* para cada algoritmo. Esto pues las diferencias respecto al *dataset* anterior son mínimas. Consideramos, por tanto, que es mucho más interesante que solo mostremos las diferencias con respecto lo desarrollado previamente.

Un **problema fundamental** con el que nos encontramos es que tenemos un problema de clasificación multiclase. Se nos presentan las siguientes posibles soluciones:

1. Trabajar este problema con `KNIME`. Para los algoritmos que trabajan directamente con clasificación multiclase, solo hace falta recopilar las métricas y curvas ROC clase por clase. En los algoritmos que no trabajan con multiclase, realizar alguna adaptación (como *One vs All*) usando *loops*
2. Realizar una adaptación incorrecta pero que simplifica nuestro trabajo. Colapsar las cuatro clases en solo dos clases. Es decir, los datos de la clase 2 tomarlos como datos de la clase 1, y los datos de la clase 4 tomarlos como datos de la clase 3. Con ello tenemos un problema de clasificación binaria que ya hemos trabajado.
3. Trabajar con `KNIME` como lo hemos hecho antes, y solo considerar la clase 1 como clase positiva. Intentar usar directamente los algoritmos sin realizar adaptaciones, y descartar aquellos que no funcionen.

En un primer momento, consideramos la primera posibilidad por ser la más correcta. Sin embargo, suponía demasiada carga de trabajo trabajar las tablas en `KNIME`, y no nos daba tiempo. Y más fundamentalmente, no hemos encontrado de realizar la adaptación *One vs All* en `KNIME`. Se puede hacer con los nodos de voto y haciendo *loops*, pero en ningún caso hemos conseguido que esto funcione.

La segunda opción la hemos descartado. Quizá sea más correcta (dentro de la gran simplificación que hacemos del problema) que la tercera. Sin embargo, el interés de este *dataset* es precisamente trabajar con un problema multiclase, y con esa aproximación perdemos la oportunidad de realizar todos los análisis respecto a este aspecto.

La tercera opción, que es la que hemos seguido, dista mucho de ser perfecta. Sin embargo, nos permite extraer conclusiones sobre el factor multiclase. Indicaremos así las carencias que tiene `KNIME` para trabajar de forma eficiente con problemas multiclase, y los problemas que tienen ciertos algoritmos con clasificación que no sea binaria. Además, también podremos discutir sobre la información que nos dan ciertas métricas respecto a tener más de una clase.

Por tanto, al seguir este proceder, no hemos realizado ningún cambio sustancial en los *workflows* que ya hemos mostrado en *"2.2.1.* Workflows *empleados para* Cross Validation*"*.

2.3.2. Mostrando los resultados obtenidos

En primer lugar, mostramos la tabla que resume todos los resultados obtenidos tras el proceso de *Cross Validation*:

Algo Name	TP	FP	TN	FN	Recall	Precision	Sensitivity	Specificity	F-measure	AUC	Accuracy	G-Mean
Naive Bayes	283	90	1064	86	0.767	0.759	0.767	0.922	0.763	0.582	0.685	0.841
Neural Net	0	789	734	0	??	0	??	0.482	??	0.995	0	??
Support Vector Machine	0	0	1145	378	0	??	0	1	??	0.488	0.247	0
Support Vector Machine + Normalization	42	28	1117	336	0.111	0.600	0.111	0.976	0.188	0.784	0.557	0.329
K-NN	275	118	1036	94	0.745	0.700	0.745	0.898	0.722	0.599	0.734	0.818
AdaBoost	204	537	608	174	0.540	0.275	0.540	0.531	0.365	0.719	0.234	0.535
Random Forest	266	126	1028	103	0.721	0.679	0.721	0.891	0.699	0.490	0.677	0.801

Cuadro 3: Resultados de *Cross Validation*, para los distintos algoritmos estudiados, en el segundo *dataset*

Notar que hay valores marcados con *??*. Ciertos algoritmos no han clasificado ningún valor en la clase positiva o ningún valor en la clase no positiva. Por tanto, en algunas métricas se divide por cero, lo que provoca estos valores *??*.

Mostramos ahora una tabla con las medidas de complejidad de cada uno de los algoritmos estudiados:

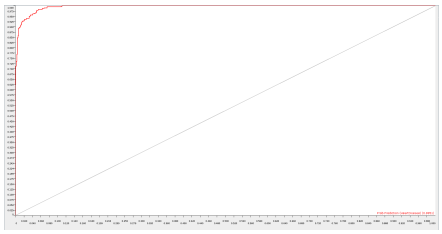
Algo Name	Medida de Complejidad
Naive Bayes	20 valores únicos por atributo
Neural Net	1 hidden layer, 10 neuronas
Support Vector Machine	Kernel RBF $\sigma = 3.0$
Support Vector Machine + Normalization	Kernel RBF $\sigma = 2.0$
K-NN	5 vecinos
AdaBoost	10 iteraciones
Random Forest	50 árboles, profundidad no acotada

Cuadro 4: Medidas de complejidad de los algoritmos considerados, en el segundo *dataset*

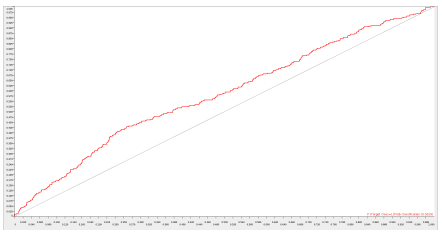
En este caso hemos modificado el valor de σ para *SVM* no normalizado y el número de vecinos para *K-NN*, respecto el primer *dataset*.

2.3.3. Mostrando las curvas ROC obtenidas

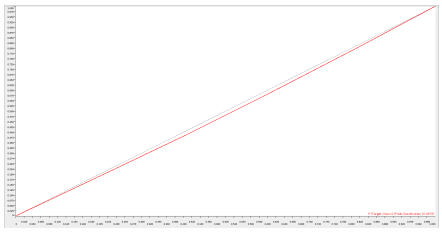
Por la decisión tomada en *"2.3.1.* Workflows *empleados para* Cross Validation*"*, tenemos algoritmos con resultados catastróficos. Esto hace que la curva *ROC* conjunta muestre comportamiento muy malos, y por ello, no podemos extraer información útil a partir de dicha gráfica conjunta. Por tanto, mostramos directamente las curvas *ROC* individuales:



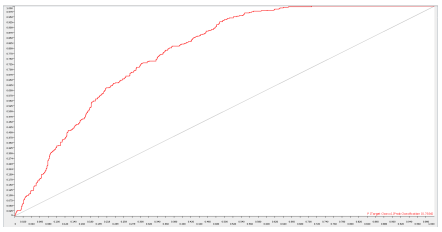
(a) Curva ROC para Redes Neuronales



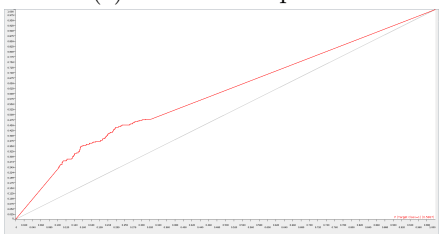
(b) Curva ROC para Naive Bayes



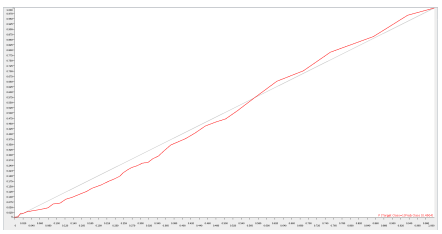
(c) Curva ROC para SVM



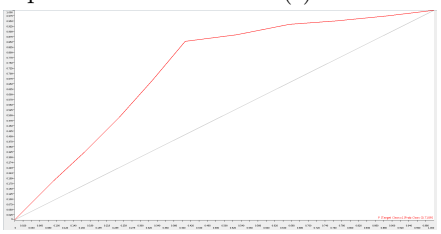
(d) Curva ROC para SVM con normalización



(e) Curva ROC para K-NN



(f) Curva ROC para Random Fores



(g) Curva ROC para Adaboost

Figura 37: Todas las curvas ROC para el segundo *dataset*, mostradas de forma individual

3. Análisis de resultados

En la anterior sección, *"2. Resultados Obtenidos"*, hemos mostrados los resultados obtenidos. Estos resultados han sido las tablas con las distintas métricas recolectadas, las medidas de complejidad de los algoritmos, y las distintas curvas ROC obtenidas. En base a esto, y a la siguiente tabla comparativa, realizaremos un análisis de los algoritmos obtenidos:

4. Configuración de los algoritmos

4.1. Consideraciones iniciales

Por cada *dataset*, hemos seleccionado dos de los mejores modelos para ese problema, y hemos realizado un ajuste de los parámetros de forma iterativa buscando la máxima eficacia de los modelos considerados. Por tanto, desarrollaremos una sección para cada *dataset* en el que realizaremos este estudio. Añadiremos también una sección para desarrollar algunas conclusiones finales sobre todos los ajustes realizados.

Observar que estamos optimizando la métrica *accuracy*, que como sabemos de teoría, tiene ciertos problemas y no es del todo representativa de la eficacia real de los modelos obtenidos. Sin embargo, esto ha sido así porque, para el nodo *Parameter Optimization Loop End*, tenemos que pasar una métrica por *flow variable*, como es posible hacer con el *scorer*. Sin embargo esto no ha sido posible usando un nodo *math formula*, por lo que nos hemos conformado con ajustar el *accuracy* (mala métrica) a través de un proceso iterativo metódico (buena forma de proceder). En otro *software* sería sencillo cambiar esta optimización para ajustar una mejor medida como *G-Mean* siguiendo la misma filosofía.

En este caso, no estamos usando *Cross Validation* para obtener las métricas. En su lugar, estamos dividiendo el conjunto de datos en 2 particiones, una para *training* (70 %) y otra para *test* (30 %). Con esto calculamos las métricas sobre el conjunto de *test*. Esta decisión ha sido tomada con el objetivo de reducir los tiempos de cómputo, que ya son elevados por la forma de proceder modificando los valores de los parámetros de forma iterativa.

Al igual que hemos hecho previamente, vamos a ser completamente explícitos en *"4.2.* Heart Failure Prediction*"*, y en los siguientes *datasets*, para evitar ser repetitivos, solo incluiremos las diferencias respecto a lo desarrollado anteriormente y los resultados concretos obtenidos para ese *dataset*. Por ejemplo, solo mostramos todas las capturas de los *workflows* en *"4.2.* Heart Failure Prediction*"*.

4.2. *Heart Failure Prediction*

Para este *dataset*, elegimos optimizar *Random Forest* y la red neuronal simple. Mostramos el *workflow* global en el que ajustamos los dos modelos:

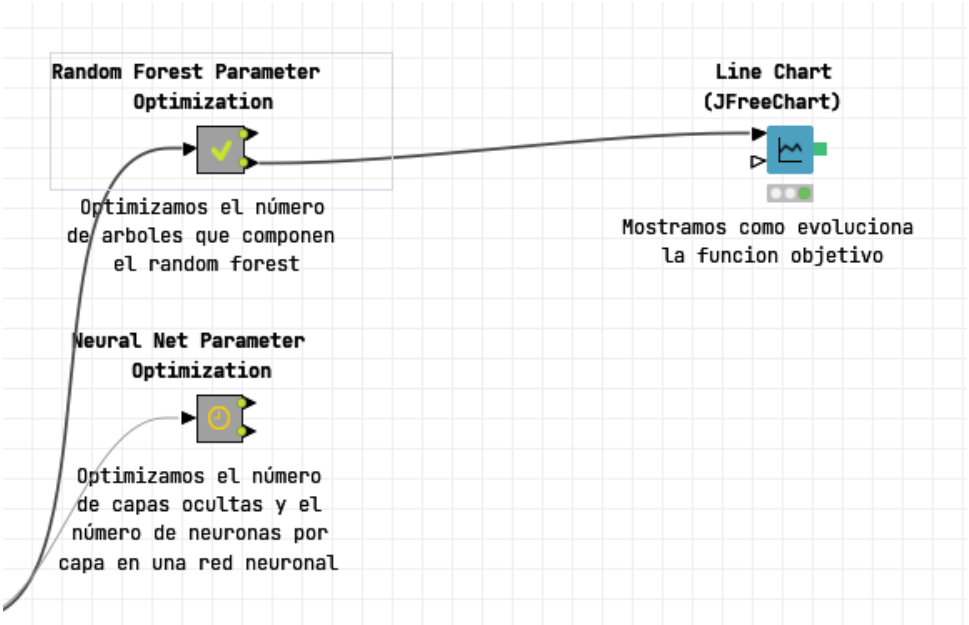


Figura 38: *Workflow* global para ajustar los dos algoritmos elegidos

Ahora mostramos la optimización que realizamos, en cada uno de los dos modelos:

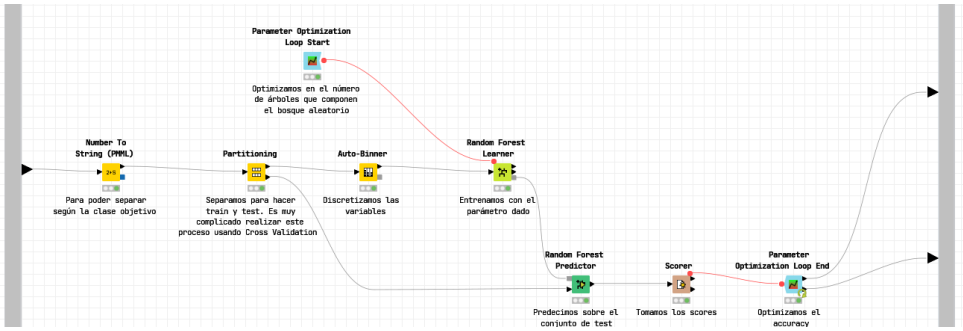


Figura 39: *Workflow* para ajustar *Random Forest*

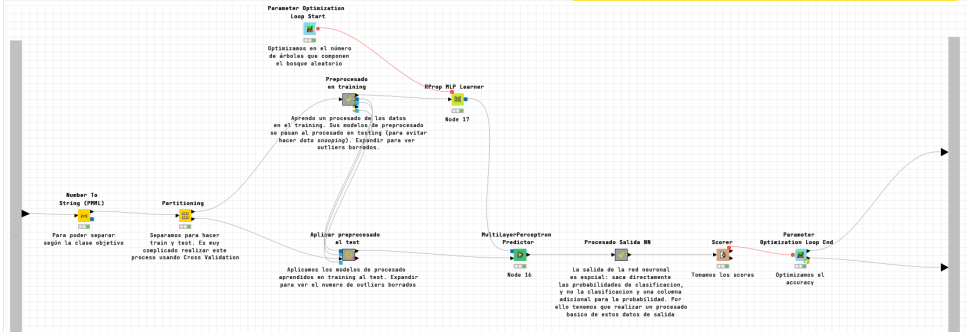


Figura 40: *Workflow* para ajustar Redes Neuronales Simples

En ambos casos estamos usando el nodo `Parameter Optimization Loop`. Para trabajar con este nodo, especificamos los parámetros que queremos optimizar, los rangos y los *steps* que tomar. Además, optimizamos usando la opción de fuerza bruta, pues estamos probando con pocas combinaciones de parámetros, por lo que nos podemos permitir este proceder. Una vez hecho esto, especificamos a nuestro modelo que use en cada iteración estos parámetros, usando para ello *flow variables*.

Una vez mostrado esto, a continuación tenemos las tablas obtenidas del proceso de optimización:

Number of trees	Objective value
10	0.837
20	0.841
30	0.859
40	0.862
50	0.859
60	0.848
70	0.851
80	0.855
90	0.859
100	0.862
110	0.862
120	0.859
130	0.862
140	0.862
150	**0.866**

Cuadro 5: Optimización para *Random Forest*, ajustando el número de árboles

En este caso, vemos que el *accuracy* crece de forma más o menos monótona conforme aumentamos el número de árboles que conforman el bosque. Por tanto, obtenemos el mejor resultado en el valor más alto de número de árboles, y lo lógico sería continuar aumentando dicho valor y ver cómo evoluciona la métrica estudiada. Pero como ya hemos comentado, estamos usando como métrica el *accuracy*, que no es una métrica del todo fiable, y por tanto decidimos no aumentar este valor en *pro* de obtener un modelo más simple y, por tanto, potencialmente mucho más generalizable.

Neurons per h.layer	Hidden Layers	Objective value
10	1	0.446
20	1	0.446
30	**1**	**0.554**
40	1	0.446
50	1	0.511
60	1	0.522
70	1	0.551
80	1	0.464
90	1	0.446
100	1	0.554
10	2	0.554
20	2	0.551
30	2	0.554
40	2	0.446
50	2	0.446
60	2	0.446
70	2	0.406
80	2	0.554
90	2	0.388
100	2	0.554
10	3	0.446
20	3	0.493
30	3	0.446
40	3	0.554
50	3	0.446
60	3	0.554
70	3	0.471
80	3	0.554
90	3	0.446
100	3	0.446

Cuadro 6: Optimización para *Redes Neuronales Simples*, ajustando el número de capas ocultas y el número de neuronas por capa oculta

En este caso, obtenemos los mejores resultados para 30 neuronas y una única capa oculta. Esto lo que nos muestra es que los mejores resultados no los estamos obteniendo con un modelo más potente, sino con uno más simple con menor capacidad de sobreajustar.

Hay que tener en cuenta que las redes neuronales de `KNIME` son muy simples, y dan muy poco de sí. Para poder trabajar con modelos más potentes de redes neuronales (más capas ocultas con más neuronas por capa), necesitamos **técnicas de regularización**. Por ejemplo, no podemos establecer una regularización tipo *weight decay*. Tampoco podemos usar *dropout* para paliar el sobreajuste. Y con ello, es lógico que modelos más simples generen mejores resultados en *test*, por su menor capacidad de sobreajuste.

Además, por problemas con el nodo `Parameter Optimization Loop`, no podemos tratar los *missing values* ni normalizar las variables sin que esto provoque fallos que no sabemos resolver. Por esto, obtenemos resultados peores que los mostrados previamente para *Custom Cross Validation*. Sin embargo, la metodología y el ajuste es correcto. Sería cuestión de encon-

trar el fallo que nos da KNIME y solucionarlo, o emplear otro *Software* para este ajuste.

4.3. Mobile Price Classification

Para este *dataset*, elegimos optimizar *Random Forest* por los buenos resultados que nos ha dado, y *K-NN*. Tanto el *workflow* global como el *workflow* para *Random Forest* no han cambiado sustancialmente, así que no mostramos capturas de pantalla para estos dos *workflows*. El *workflow* para *K-NN* se muestra en la siguiente figura:

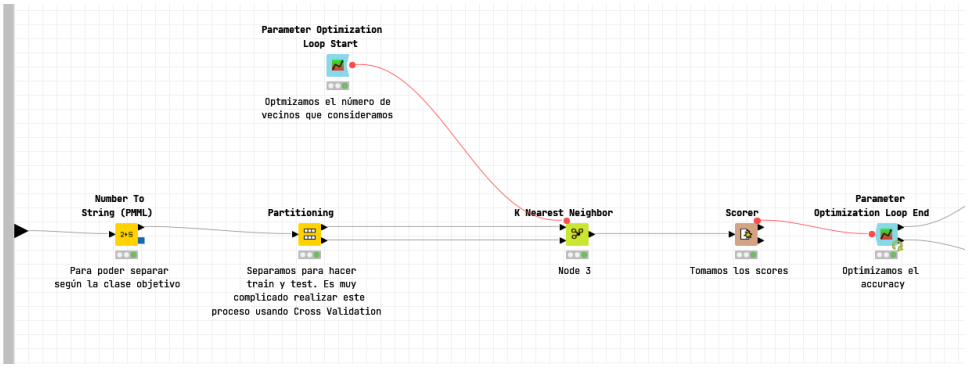


Figura 41: *Workflow* para ajustar *K-NN*

Mostramos la tabla obtenida en la optimización para *Random Forest*:

Number of trees	Objective value
10	0.805
20	0.843
30	0.872
40	**0.877**
50	0.870
60	0.868
70	0.867
80	0.868
90	0.865
100	0.875
110	0.873
120	0.875
130	0.873
140	0.868
150	0.870

Cuadro 7: Optimización para *Random Forest*, ajustando el número de árboles

Obtenemos los mejores resultados con una cantidad de árboles por cada *forest* modesta, por debajo del valor por defecto que nos da KNIME. A diferencia del *dataset* anterior, esta vez no tenemos ese crecimiento monótono en el *accuracy*, y para este caso, queda claro que en ocasiones es mejor un modelo más simple, por tanto menos potente, pero con mejor capacidad de generalización.

Mostramos ahora los resultados para la optimización de *K-NN*:

Number of neighbours	Objective value
1	0.908
2	0.908
3	0.923
4	0.925
5	0.933
6	0.937
7	0.945
8	0.940
9	0.945
10	0.942
11	0.943
12	0.947
13	0.947
14	**0.948**
15	0.947

Cuadro 8: Optimización para *K-NN*, ajustando el número de vecinos considerados para la clasificación

En este caso, estamos ajustando el número de vecinos considerados. Al aumentar este parámetro, no estamos aumentando la complejidad del modelo. Estamos aumentando la regularización que aplica al modelo. Al considerar más vecinos, suavizamos las fronteras generadas, y por tanto reducimos algo la capacidad de sobreajuste de nuestro modelo.

Con ello, estamos viendo que nuestro modelo se beneficia enormemente de la regularización aplicada. Regularización que, en general, hemos echado de menos en las herramientas que `KNIME` pone a nuestra disposición.

Además, cabe resaltar los buenos resultados que obtenemos en *accuracy*. Al tener cuatro clases perfectamente balanceadas, este buen *accuracy* no puede explicarse por un clasificado constante o algún problema de esta índole. Y por tanto, podemos asegurar que estamos obteniendo un modelo de muy buen rendimiento.

4.4. Conclusiones finales

5. Procesado de datos

5.1. Consideraciones iniciales

Desarrollaremos distintos apartados para cada uno de los *datasets*. En vista de los análisis previos para cada *dataset*, tomaremos ciertas decisiones de procesado buscando mejorar el rendimiento de los modelos planteados. Compararemos dicho rendimiento antes y después del procesado, lo que nos permitirá extraer ciertas conclusiones sobre la técnica empleada, sobre el comportamiento de los algoritmos o sobre el problema en sí.

5.2. *Heart Failure Prediction*

Como hemos estudiado en *"5. Matriz de correlaciones lineales. Un color azul significa correlación lineal positiva, y un color rojo significa correlación lineal negativa."*, tenemos variables que están muy correladas entre sí. Así que para mejorar este *dataset*, decidimos borrar algunas variables que estén altamente correladas con otras, para ver si conseguimos obtener mejores resultados o, al menos, mantener el mismo rendimiento con menos variables, y por tanto, con modelos más sencillos.

Podríamos haber aplicado otras técnicas. *PCA* ya se demostró poco efectivo (al menos con dos dimensiones) en el análisis exploratorio de datos, pues no proporcionaba una buena distribución de datos. Además tenemos muy pocas variables como para necesitar un método automático de reducción de dimensionalidad (es factible seleccionar las variables manualmente). Tampoco tenemos un desbalanceo excesivo de las clases como para aplicar una técnica como *SMOTE*. Son estas las razones por las que nos decantamos por un procesado de los datos tan básico de los datos (que ya de por sí son básicos).

Este procesado de los datos se realiza del siguiente modo:

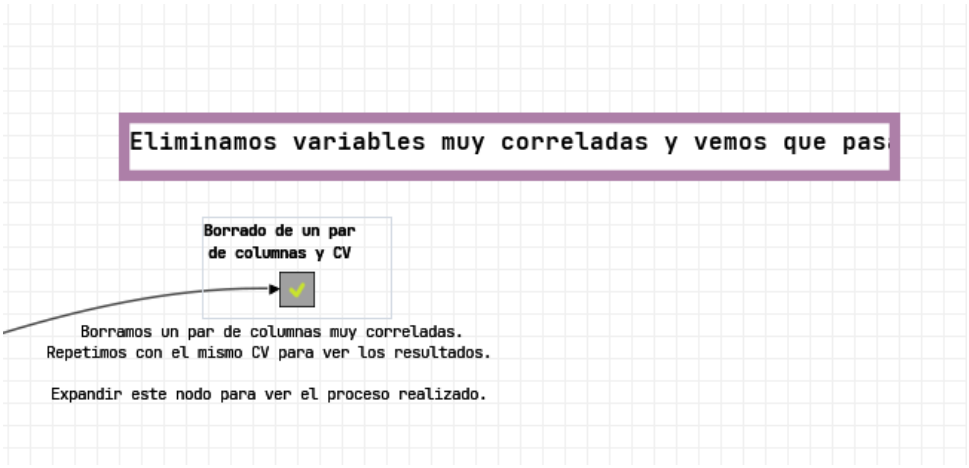


Figura 42: *Workflow* de más alto nivel para el procesado de los datos

Mostramos el contenido de este nodo:

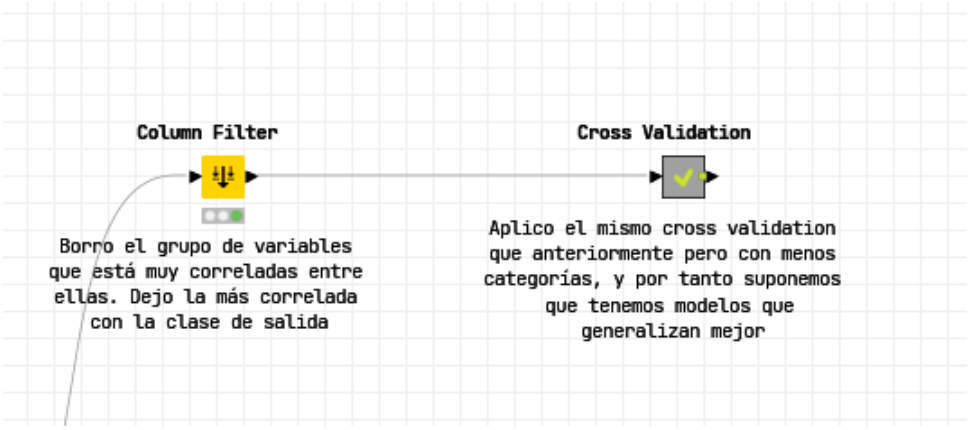


Figura 43: *Workflow* de más alto nivel para el procesado de los datos

En este nodo nos quedamos solo con las variables `MaxHR`, `ExerciseAngina`, `Oldpeak`, `ST_Slope` y la variable de salida (decisión justificada por lo que comentábamos en el *EDA* para este *dataset*).

Además, tenemos que adaptar los nodos de *Custom Cross Validation*. Como no estamos usando la variable `Cholesterol`, no tenemos que crear los *missing values*, porque ya no tenemos. Este es el cambio que realizamos en todos los nodos de *Custom Cross Validation*.

Los resultados obtenidos tras aplicar *Cross Validation* tras esta transformación son los siguientes:

Algo Name	TruePositives	FalsePositives	TrueNegatives	FalseNegatives	Recall	Precision	Sensitivity	Specificity	F-measure	Area Under Curve	Accuracy	G-Mean
Naive Bayes	328	135	314	140	0.701	0.708	0.701	0.699	0.705	0.559	0.700	0.700
Neural Net	431	86	324	76	0.850	0.834	0.850	0.790	0.842	0.880	0.823	0.820
Support Vector Machine	421	102	308	86	0.830	0.805	0.830	0.751	0.817	1	0.795	0.790
Support Vector Machine + Normalization	429	93	317	78	0.846	0.822	0.846	0.773	0.834	1	0.814	0.809
K-NN	151	268	142	356	0.298	0.360	0.298	0.346	0.326	0.828	0.320	0.321
AdaBoost	388	87	362	80	0.829	0.817	0.829	0.806	0.823	0.565	0.818	0.818
Random Forest	314	128	321	154	0.671	0.710	0.671	0.715	0.690	0.576	0.692	0.693

Cuadro 9: Resultados de *Cross Validation*, para los distintos algoritmos estudiados, en el primer *dataset*, tras realizar la selección de variables

Aunque ya hemos mostrado esta tabla previamente, mostramos por claridad la tabla de *Cross Validation* obtenida sin el procesado de los datos, para que sea más sencillo realizar la comparación:

Algo Name	TP	FP	TN	FN	Recall	Precision	Sensitivity	Specificity	F-measure	AUC	Accuracy	G-Mean
Naive Bayes	212	106	247	79	0.73	0.67	0.73	0.70	0.70	0.92	0.71	0.71
Neural Net	212	42	179	48	0.82	0.83	0.82	0.81	0.82	0.88	0.81	0.81
Support Vector Machine	6	1	352	285	0.02	0.86	0.02	1.00	0.04	1	0.56	0.14
Support Vector Machine + Normalization	141	33	332	195	0.42	0.81	0.42	0.91	0.55	1	0.67	0.62
K-NN	155	161	192	136	0.53	0.49	0.53	0.54	0.51	0.72	0.54	0.54
AdaBoost	208	102	251	83	0.71	0.67	0.71	0.71	0.69	0.92	0.71	0.71
Random Forest	215	107	246	76	0.74	0.67	0.74	0.70	0.70	0.92	0.72	0.72

Cuadro 10: Resultados de *Cross Validation*, para los distintos algoritmos estudiados, en el primer *dataset*, **sin aplicar el procesado de datos**

5.2.1. Conclusiones

En base a las dos tablas de resultados anteriores, *"9. Resultados de* Cross Validation*, para los distintos algoritmos estudiados, en el primer* dataset*, tras realizar la selección de variables"* y *"10. Resultados de* Cross Validation*, para los distintos algoritmos estudiados, en el primer* dataset*,* ***sin aplicar el procesado de datos****"*, realizamos el siguiente análisis.

Nos fijaremos en la métrica *G-Mean*, pues es más informativa que le *accuracy*, como ya hemos señalado varias veces a lo largo de esta memoria. En general hemos mejorado significativamente esta métrica quedándonos con menos columnas en comparación a los resultados sin este preprocesado de datos. Esto es claro pues pasamos de tener un *G-Mean* medio de 0.6071 a tener un *G-Mean* medio de 0.7072. Por tanto, **mejoramos el rendimiento generando modelos más sencillos**.

Nos fijamos ahora en las diferencias algoritmo a algoritmo:

Algo Name	$procesado - sin\_procesar$
Naive Bayes	-0.010
Neural Net	0.010
SVM	0.650
SVM Norm	0.189
KNN	-0.219
Adaboost	0.108
Random Forest	-0.027

Cuadro 11: Tabla con las diferencias algoritmo a algoritmo, rendimiento del algoritmo tras el procesado menos el rendimiento algoritmo sin el procesado. El rendimiento es *G-Mean*

Cuando el algoritmo pierde rendimiento, este empeoramiento es muy pequeño (la peor diferencia es ≈ -0.22). Por tanto podemos decir que, en esos casos estamos manteniendo el rendimiento con modelos más sencillos. Las mejoras, sin embargo, son más grandes (prácticamente el triple de grandes que los empeoramientos, llegando a ≈ 0.65). Y todo esto teniendo en cuenta que estamos trabajando con un *dataset* mucho más pequeño y, por tanto, generando modelos mucho más simples, por tanto mucho más generalizables y, cuando el algoritmo lo permite, modelos muchos más interpretables.

El algoritmo más perjudicado ha sido *K-NN*. Esto puede ser debido a que, a diferencia de lo que estamos haciendo más adelante, no estamos normalizando los datos de entrada. Por tanto es más sensible a diferencias de escala en las variables. Y al tener menos variables, este efecto es mucho más crítico. Además, al ser un algoritmo que considera información de forma local, al estar quitando variables en las que mirar en la vecindad, es un algoritmo que pierde mucha información por este procesado.

El algoritmo más beneficiado ha sido *SVM* sin normalización. Es destacable que *SVM* con normalización no se haya beneficiado tanto. Por tanto, y como comentábamos para *K-NN*, las distintas escalas de las variables han tenido mucho que ver en este resultado. Creemos que *SVM* sin normalización estaba sufriendo demasiado por diferencias en escala de alguna de las variables que hemos eliminado. Al eliminar estas variables problemáticas, hemos conseguido que el algoritmo tenga mucho mayor rendimiento.

A pesar de esto último, no olvidar que *SVM* con normalización también se ve beneficiado

del procesado. Por tanto, además de haber eliminado variables problemáticas para *SVM* sin normalizar, el procesado es en sí mismo beneficioso para el rendimiento de estos modelos.

5.3. Mobile Price Classification

Como hemos comentado en *"1.3.1. Análisis Exploratorio de los datos"*, aplicar *PCA* resultando en solo dos variables de entrada parecía muy prometedor. Así que hemos aplicado esto a nuestro *dataset*, y luego hemos comparado con el rendimiento de un algoritmo en concreto.

La estructura general se muestra en el siguiente *workflow*:

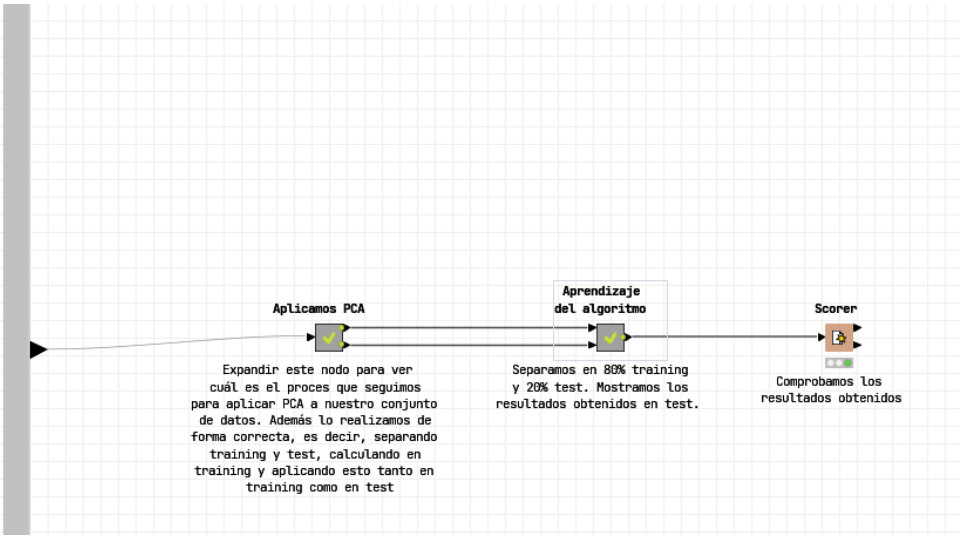


Figura 44: *Workflow* general aplicado para el procesado de datos

Aplicamos *PCA* de la forma correcta. Para ello separamos en *training* y *test*. Realizamos los cálculos de *PCA* sobre *training*, y los aplicamos a *training*. Con esos cálculos, aplicamos *PCA* a test. Esto es importante para evitar hacer *data snooping*. Esto se muestra en el siguiente *workflow*:

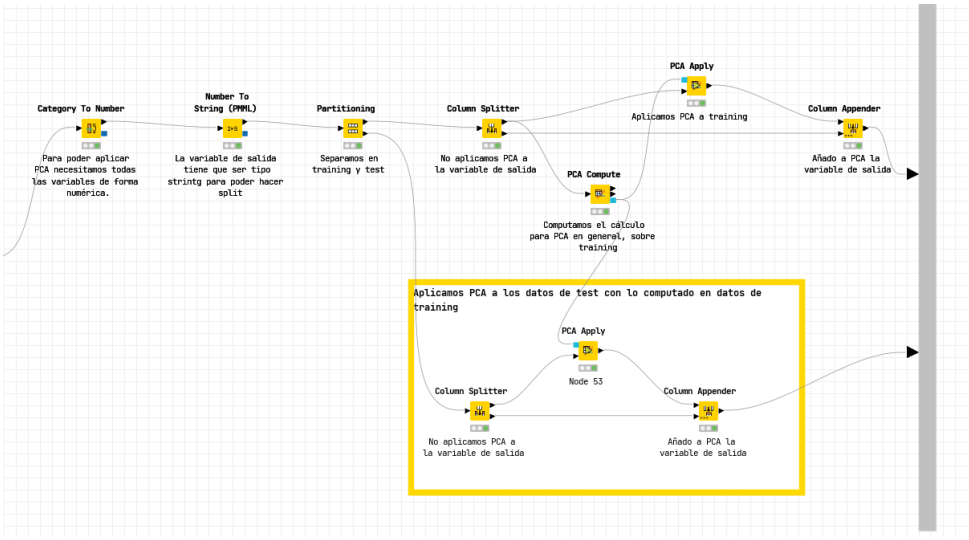


Figura 45: *Workflow* para aplicar *PCA* de forma correcta

Con esto, usamos *k-NN* para realizar las predicciones. Ya hemos visto que este algoritmo bien empleado puede obtener resultados realmente buenos. Mostramos este *workflow*:

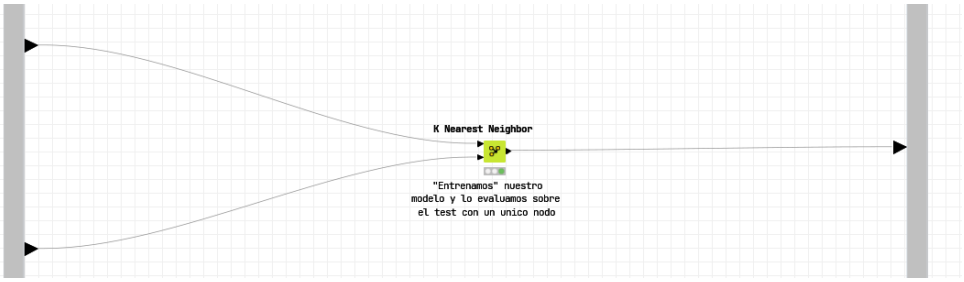


Figura 46: Evaluando el procesado de los datos usando un modelo de aprendizaje automático concreto

Los resultados obtenidos del *scorer* se muestran en la siguiente tabla:

Classification Class	TruePositives	FalsePositives	TrueNegatives	FalseNegatives	Recall	Precision	Sensitivity	Specificity	F-measure	Accuracy	Cohen's kappa
1	116	41	409	34	0.773	0.739	0.773	0.909	0.756		
2	109	38	412	41	0.727	0.741	0.727	0.916	0.734		
3	134	18	432	16	0.893	0.882	0.893	0.960	0.887		
0	132	12	438	18	0.880	0.917	0.880	0.973	0.898		
Overall										0.818	0.758

Cuadro 12: Resultados de entrenar *K-NN*, con 10 vecinos, en el conjunto de datos tras el procesado

5.3.1. Conclusiones

En *"8. Optimización para* K-NN*, ajustando el número de vecinos considerados para la clasificación"* conseguíamos, tras un ajuste, un *accuracy* de ≈ 0.94. Tras este procesado, y sin ajuste alguno, obtenemos un *accuracy* de 0.818.

Esto es realmente sorprendente por dos motivos. El primero y más obvio es que estamos reduciendo toda la información del *dataset* a solamente dos variables. De 20 variables de entradad hemos pasado a solamente utilizar 2 variables, una reducción $x10$, lo cual de por sí ya es muy relevante. El segundo motivo es que estamos obteniendo resultados realmente buenos con un modelo realmente simple, como es *K-NN*.

Estamos perdiendo aproximadamente un 0.13 de *accuracy* para el caso concreto de *K-NN*. Sin embargo, estamos simplificando el problema de forma masiva manteniendo un *accuracy* muy decente. El único problema que le vemos a este procedimiento es que perdemos la interpretabilidad en el momento en que colapsamos todo el *dataset* a dos variables continuas. Sin embargo, la simplificación es innegable. Y lo que ello conlleva: modelos más rápidos de entrenar, más rápidos en inferencia y mucho más generalizables.

6. Interpretación de los resultados

6.1. Consideraciones iniciales

Al igual que en otras secciones, desarrollaremos una interpretación de los resultados por cada uno de los problemas enfrentados. Además, realizaremos una interpretación integrando todas las conclusiones individuales desarrolladas para cada uno de los *datasets*.

6.2. *Heart Failure Prediction*

La observación más importante y directa para este problema, tras el estudio que hemos desarrollado de este, es la relevancia de las variables que componen el *dataset*. En *"5.2.* Heart Failure Prediction*"* ha quedado claro que solo con un subconjunto de variables, muy correladas entre sí pero sobre todo correladas con la variable de salida, es más que suficiente para generar modelos con buen rendimiento. Es más, no es que haya sido suficiente, sino que ha sido mejor que considerar todo el conjunto de variables, como se ha demostrado en *"5.2.1. Conclusiones"*.

Como comentábamos previamente, las variables fundamentales que han aportado todo el conocimiento necesario para generar los modelos predictivos han sido `MaxHR`, `ExerciseAngina`, `Oldpeak`, `ST_Slope`.

Para ver la relevancia de estas variables, entrenamos un árbol de decisión con el *dataset* procesado y mostramos dicho árbol de forma gráfica, en la siguiente figura:

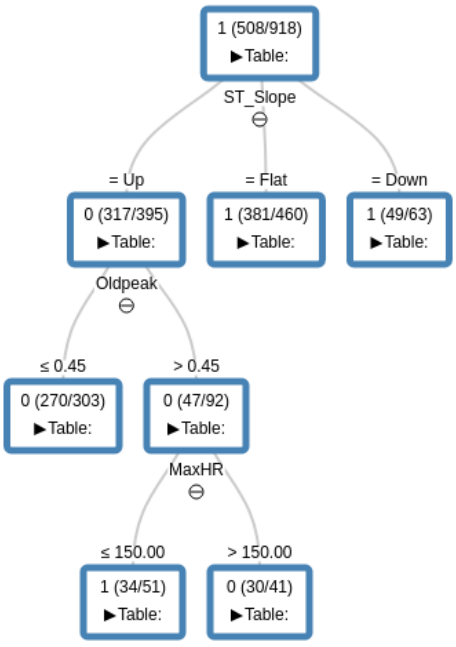


Figura 47: *decision tree* entrenado sobre todo el *dataset* tras aplicar el procesado anteriormente descrito

Con esto queda clara la relevancia de las variables. En primer lugar tenemos a la variable `ST_Slope`, que es la que permite una mayor división del *dataset* en *subdatasets* con menor diversidad. En segundo lugar, tenemos la variable `Oldpeak`, y para terminar la variable `MaxHR`

.

Esta visualización la realizamos añadiendo lo siguiente al nodo de procesado de datos, que hemos mostrado previamente en esta memoria de prácticas:

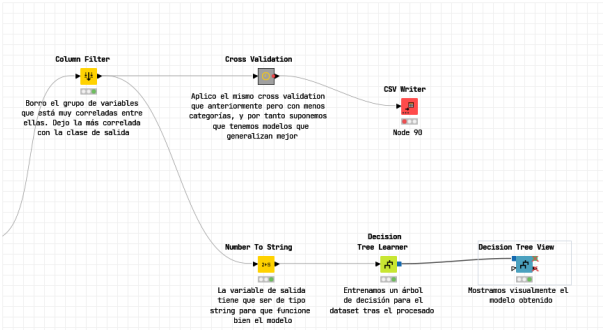


Figura 48: Mostramos el añadido al nodo de procesado de datos, para poder mostrar gráficamente el árbol de decisión que nos da información sobre la relevancia de las variables

En segundo lugar, ha sido crítico percatarnos de la existencia de *missing values* que KNIME no ha detectado. Esto nos ha permitido ignorar ciertas filas en el proceso de entrenamiento. Si no hubiésemos hecho esto, habríamos obtenido un modelo que explota datos sobre *missing values* de forma *"tramposa"*, produciendo un modelo mucho más frágil y por tanto, menos generalizable. Esto porque hacer esto es equivalente a seleccionar los *missing values* y tratarlos asignando una variable constante arbitraria. Hemos visto en la teoría de esta asignatura que esta forma de proceder no es buena.

En definitiva, un dataset que ya de por sí era sencillo (solo 11 variables de entrada), hemos conseguido trabajarlo para emplear todavía menos variables conservando (es más, mejorando) las medidas de rendimiento de nuestros modelos generados.

6.3. Mobile Price Classification

Como pasaba en el anterior *dataset*, la observación más directa y relevante viene a partir del procesamiento de datos. Hemos conseguido reducir un problema con 20 variables de entrada a un problema con 2 variables de entrada empleando *PCA*, y con un modelo de aprendizaje automático simple como *K-NN*. Como ya hemos mencionado en *"5.3.1. Conclusiones"*, esto es relevante por dos motivos. El primero, porque estamos reduciendo un problema de 20 variables de entrada a solo dos variables. Con ello, estamos reduciendo la estructura del problema a tratar a una mucho más sencilla, y con ello podremos entrenar e inferir modelos de una forma mucho más rápida, con una capacidad de generalización mucho más alta. En segundo lugar, es relevante que un modelo simple como *K-NN* logre unos resultados tan buenos, simplemente regularizando adecuadamente el modelo entrenado.

Como hacíamos en el primer *dataset*, exploramos algo más la estructura del problema usando árboles de decisión. Mostramos los primeros niveles, que nos muestran las variables más relevantes (pues producen las subdivisiones con menor diversidad) en nuestro *dataset*:

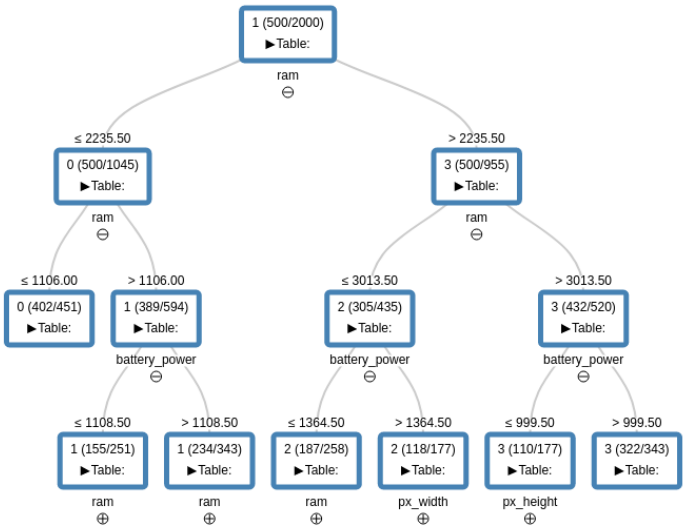


Figura 49: Árbol de decisión construido sobre todo el conjunto de datos. Solo mostramos los primeros niveles

Vemos que claraemnte la variable `ram` tiene un peso enorme a la hora de construir el clasificador. Esto ya lo sospechábamos por lo que hemos comentado en *"12. Matriz de correlaciones lineales. Un color azul significa correlación lineal positiva, y un color rojo significa correlación lineal negativa"*.

Por tanto, hemos comenzado con un *dataset* relativamente complicado, conteniendo 20 variables. Y trabajando los datos y resultados obtenidos, hemos logrado conseguir reducir el problema a uno brutalmente más sencillo, en el que usando 2 variables hemos obtenido resultados muy buenos. Además, hemos escogido reducirlo a solo 2 variables por la facilidad que otorga en la visualización. Si usásemos 3 variables, seguramente obtendríamos unos modelos con un rendimiento todavía mucho más bueno, manteniendo una gran simplicidad en el modelo.

6.4. Interpretación global de todos los problemas estudiados

7. Referencias

[1] "Heart failure prediction dataset — kaggle." https://www.kaggle.com/fedesoriano/heart-failure-prediction. (Accessed on 11/05/2021).

[2] "A step-by-step explanation of principal component analysis (pca) — built in." https://builtin.com/data-science/step-step-explanation-principal-component-analysis. (Accessed on 11/05/2021).

[3] "Mobile price classification — kaggle." https://www.kaggle.com/iabhishekofficial/mobile-price-classification. (Accessed on 11/07/2021).

[4] "Distribución normal - wikipedia, la enciclopedia libre." https://es.wikipedia.org/wiki/Distribuci%C3%B3n_normal#Desviaci%C3%B3n_t%C3%ADpica_e_intervalos_de_confianza. (Accessed on 11/06/2021).