



UNIVERSIDAD DE GRANADA

INTELIGENCIA DE NEGOCIO
GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 3

COMPETICIÓN EN KAGGLE

Autor

José María Sánchez Guerrero






















Rama

Sistemas de Información



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

CURSO 2020-2021

Overview	Data	Notebooks	Discussion	Leaderboard	Rules	Team	My Submissions	Late Submission
1	—	JuanHeliosGarcía					 0.83002	15 3d
2	—	PATRICIA CORDOBA 77145053					 0.82830	13 3d
3	—	José Alberto García 26513007X					 0.82484	43 1d
4	—	Alvaro de Rada 49108766V					 0.81622	14 18h
5	—	DAVID CABEZAS 20079906					 0.81622	34 2d
6	—	OctavioTorres					 0.81622	23 18h
7	—	AlejandroAlonso75577394S					 0.81363	10 2d
8	—	Mikhail Raudin 531101855					 0.80845	11 17h
9	—	Javier Rodríguez 78306251Z					 0.80759	12 18h
10	—	David Martín 75931868J					 0.80586	28 1d
11	—	JuanCarlosGonQu					 0.79982	54 1d
12	—	Pedro Jiménez 76592485R					 0.79810	35 1d
13	—	Ilias_Amar_Ceuta					 0.79723	15 21h
14	—	Jose Antonio Martín 77561280J					 0.79551	14 21h
15	—	Alberto_Postigo_Ceuta					 0.79206	21 2d
16	—	Laura Delgado 20608068E					 0.79119	18 20h
17	—	Jose María Sánchez Guerrero ...					 0.79119	19 19h
18	—	Sergio Fernández Fernández U...					 0.78688	12 18h
19	—	Alejandro Menor Molinero 1317...					 0.77911	4 10d
20	—	Antonio Jesús Ruiz 53911182x					 0.77825	8 18h
21	—	Daniel Perez 26513557P					 0.77653	16 1d

Índice

1. Introducción	2
2. Estrategias y progreso obtenido	2
2.1. Primeros pasos	2
2.2. Intentando mejorar los modelos	6
2.3. Cambiando el planteamiento	7
3. Tabla de soluciones	11
Referencias	12

1. Introducción

En esta última práctica pondremos a prueba lo aprendido en las prácticas anteriores a través de una competición en la plataforma Kaggle. En ella se nos plantea un problema de clasificación de una serie de coches vendidos, donde tendremos que predecir la categoría del precio del coche (del 1 al 5, yendo de más barato a más caro), por lo tanto estamos ante un problema de clasificación multiclase.

Dispondremos de un conjunto de entrenamiento, con 4819 ejemplos de coches vendidos y el precio con el que fueron vendidos; y un conjunto de test, con 1158 ejemplos de coches vendidos y los cuales tendremos que predecir el precio al que se vendieron. Este resultado será el que subiremos a la web para comprobar que tal ha funcionado la estrategia que hemos seguido. Para medir el rendimiento se utilizará la precisión (*accuracy*) obtenida.

2. Estrategias y progreso obtenido

Como no se ha seguido una estrategia planeada, si no que se ha ido modificando dependiendo de los resultados obtenidos, vamos a explicar poco a poco que métodos hemos utilizado y cómo los hemos mejorado. En algunos casos no se ha podido mejorar, no obstante, también explicaremos que se había intentado y por qué.

2.1. Primeros pasos

En primer lugar, intentamos comprender los datos, ver con qué estamos trabajando y explorar los distintos métodos que tenemos a nuestra disposición para resolverlo. Se utiliza un sólo *script* para esta parte (*'primeros-pasos.ipynb'*), ya que entre unas ejecuciones y otras se cambiaban unos parámetros, o bien, se ejecutan a la vez.

Para empezar, nada más leer el dataset, vemos que hay bastantes datos que faltan o datos nulos. Unos 72 o más por atributo, y hasta 4160 de 'Descuento'. Este último, es más lógico pensar que si no tenemos un dato del descuento, es porque no ha habido ninguno, así que estos datos los podremos rellenar con un 0. Puede ser que algún dato tuviese un descuento de verdad, y que no fuese nulo, sin embargo, es más complicado de prever además de que es un valor realista (no como por ejemplo, un valor de 0 en 'Motor_CC'). Para el resto de datos, he intentado rellenarlos con algo de lógica. Las funciones de *Pandas* *'ffill()'* y *'bfill()'* rellenan el dato en blanco con el que hay justo al lado; así que he ordenado los datos por nombre (marca y modelo) y así la posibilidad de que se rellene correctamente

son mayores, ya que un coche la potencia del motor, consumo o combustible serán iguales. Otros datos como el año o los kilómetros si que serán distintos, por lo que igual puede convenir más rellenarlo de otra forma.

Lo siguiente que hacemos es simplemente quitarle las unidades a los datos, es decir, en vez de tener en los datos de 'Potencia' un *74bhp*, ahora tendremos un *74*.

Por último, utilizaremos *LabelEncoder()* para codificar las columnas representadas con 'string' y que se muestren como enteros, ya que prácticamente todos los modelos que se utilizan, trabajan con enteros o escalados a flotantes. Una vez hecho este pre procesamiento de datos, mostramos como han quedado estos y la matriz de confusión resultante, por si podemos sacar algunas conclusiones:

Nombre	Ciudad	Año	Kilometros	Combustible	Tipo_marchas	Mano	Consumo	Motor_CC	Potencia	Asientos	Descuento	Precio_cat
0	2	2014.0	79271.0	1	0	0	20.38	1968	143	5.0	0.00	5
0	0	2016.0	20003.0	1	0	0	20.38	1968	143	5.0	0.00	5
0	0	2016.0	39000.0	1	0	0	20.38	1968	143	5.0	42.89	5
0	8	2017.0	22000.0	1	0	2	20.38	1968	143	5.0	0.00	5
1	0	2011.0	53000.0	4	0	0	12.3	1781	163.2	5.0	0.00	4



A continuación, pasamos a crear los modelos de clasificación. He introducido una pequeña sección de código que me sirve para ejecutar unas pruebas rápidas de algún clasificador. Este código también lo utilizaré en los demás scripts.

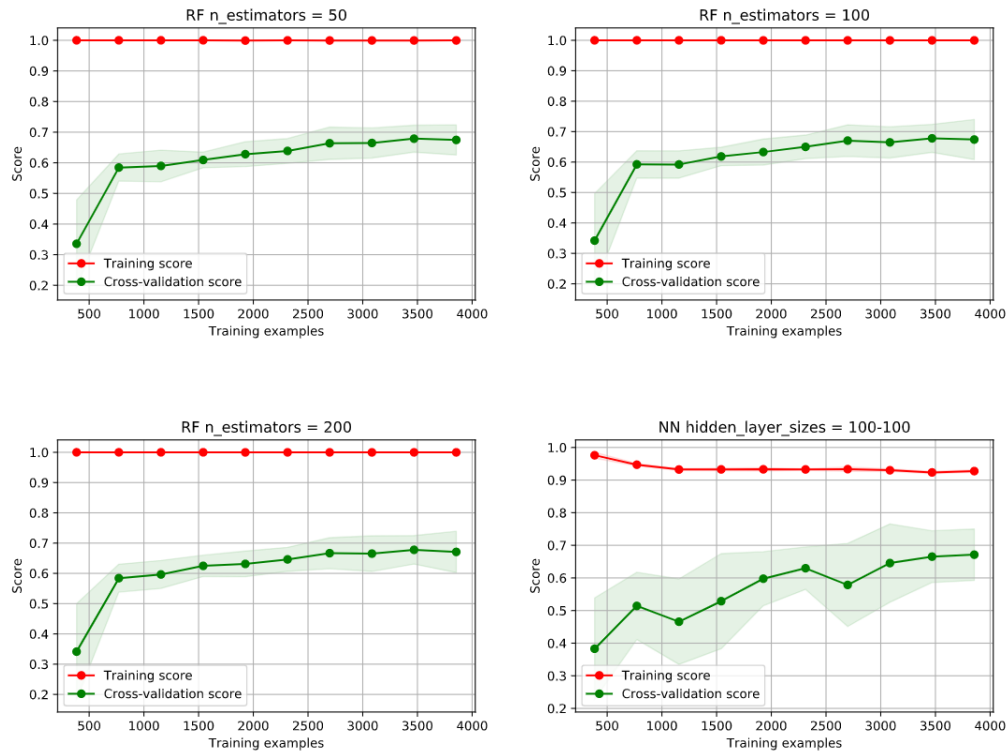
```
1  # Declaracion del modelo
2  RndForestClf = RandomForestClassifier(n_estimators=100)
3
4  # Calculo de las predicciones mediante validacion cruzada (5-folds)
5  y_pred = cross_val_predict(RndForestClf, x_train, y_train, cv=5)
6
7  print(classification_report(y_train, y_pred))
8  print("SCORE: ", accuracy_score(y_train, y_pred))
9
10 # Matriz de confusion de los resultados
11 confusion_matrix = confusion_matrix(y_train, y_pred)
12
13 sns.heatmap(confusion_matrix, annot = True, fmt='g')
```

En el código, lo que se hace es declarar y generar el modelo que vamos a testear, utilizar validación cruzada sobre el conjunto de entrenamiento para ver cómo de eficaz es, y por último, mostramos las medidas de precisión obtenidas junto a su matriz de confusión correspondiente.

Como ya he dicho, este código lo utilizaremos en scripts posteriores, sin embargo, en este primero no me fue tan útil, ya que mi intención era probar cuantos más modelos mejor y me resultaba bastante tedioso hacerlo de esta forma. Para solucionarlo, cree una lista de *pipelines*, en los cuales metía un modelo junto con los parámetros que quería probar. No tengo todos los modelos que probé, pero si tengo las últimas pruebas con los que mejores resultados obtuve (*RandomForest* y *MLPerceptron*). Este es el código con el cual los generaba:

```
1  # Crear lista de pipelines
2  pipelines = []
3
4  # Insertar nuevo pipeline
5  for hidden_layer_sizes in hidden_layer_sizes_list :
6      pipelines.append(
7          make_pipeline(
8              StandardScaler(),
9              MLPClassifier(hidden_layer_sizes=hidden_layer_sizes,
10                           early_stopping=False, random_state=1)
11          )
12      )
13  nn_pipe = pipelines
```

Finalmente, para cada uno de los modelos de la lista, evaluamos su rendimiento he imprimimos tanto la media de aciertos como la desviación típica que obtuvo cada uno. Además de esto, también se incluye una función que nos muestra una gráfica de la curva de aprendizaje de cada uno de los modelos probados, y así poder analizar mejor cómo han funcionado nuestros algoritmos.



Podemos ver las curvas de aprendizaje de los modelos que mejores resultados obtuvieron. Como se puede apreciar, no son malas pero como que el *score* obtenido en la validación cruzada se estanca sobre el 70 %. No obstante, utilizamos el dataset de entrenamiento hacer a su vez el test, así que decidí probar a entrenar con todo entero y subir los resultados a la web.

Estos van a ser los ficheros 1 y 2 adjuntados en el zip (y en la tabla de soluciones), el primero con una red neuronal y el segundo con el *random forest*.

Los resultados obtenidos no fueron los esperados, la verdad, ya que obtuve un 0.61518 para el *MLPerceptron* y un 0.72131 para el *RandomForest*. Por una parte, la puntuación de este último no me pareció muy mala, pese a que mi posición era de mitad de tabla para abajo (no recuerdo muy bien estas primeras posiciones y tampoco las anoté). Pero por otra, me sorprendió la que obtuvo la red neuronal, con un porcentaje mucho más bajo que el pronosticado y colocándome el penúltimo, si no recuerdo mal.

2.2. Intentando mejorar los modelos

Debido a los resultados obtenidos, intentamos mejorar el funcionamiento de los modelos, comenzando por el de la red neuronal. Mi primera idea fue volver a probar más parámetros del *MLPerceptron*, pero como las pruebas rápidas realizadas continuaban dando resultados similares, decidí cambiar de estrategia e implementar yo la red neuronal utilizando *Keras*:

```
1  # Build the model
2  model = Sequential()
3
4  model.add(Dense(10, input_shape=(12,), activation='relu'))
5  model.add(Dense(10, activation='relu'))
6  model.add(Dense(6, activation='sigmoid'))
7
8  # Adam optimizer with learning rate of 0.0001
9  optimizer = Adam(0.0001)
10 model.compile(optimizer, loss='categorical_crossentropy',
11               metrics=['accuracy'])
```

Los parámetros para la red también fueron elegidos mediante pruebas rápidas y esos fueron los que mejor resultado dieron. Luego, a la hora de subirlo a la web, la puntuación obtenida fue de 0.46764, lo cual ya me hizo pensar que la solución elegida no era la correcta y que debía centrarme más en modelos como *RandomForest*, *AdaBoost* o similares.

Ese mismo día probé a subir un *RandomForest* con alguna pequeña modificación y con la cual obtuve un resultado de 0.69111. Posteriormente, ya si probé a más opciones, y no sólo con parámetros de este clasificador, si no que también con los datos. La primera idea fue dejando la marca del coche (en vez de marca y modelo), cambiando el autocompletado de valores nulos o no normalizando los datos finales. Sorprendentemente, lo que mejor resultado me dió fue esto último, aunque no entiendo muy bien la razón. Igual fue por pérdida de información o porque no estaba normalizando bien; sin embargo, las tres subidas que hice ese día obtuvieron una precisión del 0.71182, 0.75323 y 0.75150¹, colocándome en la posición 20 de 30 que estaban participando en ese momento, aproximadamente.

Las siguientes subidas un poco más de lo mismo. Intento cambiar parámetros y/o detalles del preprocesamiento a ver si mejoraba, pero sin resultado alguno. Las primeras subidas del día 30 también pertenecen a estas pruebas y que, debido a las 3 subidas diarias, no pude comprobar el día anterior. Más concretamente pertenecen al *StackingClassifier*, un método nuevo que consiste en 'apilar' varios clasificadores en una salida perteneciente a un estimador final. Es decir, le pasamos varios clasificadores para calcular la predicción final, y dependiendo de la fuerza

¹De estas ejecuciones sólo conservo la que obtuvo mayor porcentaje, las otras las he debido de eliminar sin querer

que tenga cada clasificador individualmente, se utilizará su salida como entrada de un estimador final (este será en mi caso *LogisticRegression()*, ya que es el más utilizado). Este clasificador junto con los anteriormente explicados estarán en el notebook '*mejorando-modelos.ipynb*':

```
1  # Clasificadores a apilar en el modelo
2  estimators = [
3      ('rf', RandomForestClassifier(n_estimators=100)),
4      ('svr', make_pipeline(MinMaxScaler(),
5                             LinearSVC()))
6      ]
7
8
9  # Declaracion del modelo
10 StackingClf = StackingClassifier(estimators=estimators,
11                                  final_estimator=LogisticRegression())
```

Si observamos los resultados en la web, no fueron muy buenos (0.70750 y 0.71786), algo que me resultó bastante extraño. Las pruebas rápidas que hice con este modelo me dieron una precisión del 72% más o menos. Teniendo en cuenta que se hace con validación cruzada y que el conjunto con el que entrenamos se ve un poco más reducido, me resultó extraño. En este momento estaba un poco bloqueado, por lo que decidí cambiar el planteamiento totalmente y empezar de nuevo.

2.3. Cambiando el planteamiento

Viendo que mis resultados no mejoraban y que también estaba bajando bastantes posiciones en la tabla (iría ya por la posición 30-35 de ya unas 40 personas que hay), me hicieron pensar que no todo era el clasificador que estaba utilizando, sino que el preprocesado de los datos también influye bastante en el resultado final.

Me estuve leyendo las transparencias, viendo las prácticas anteriores y buscando un por internet para informarme en qué podía mejorar (sobre todo me ayudó una pequeña conferencia de la Universidad de DePaul [7]). El comienzo es muy similar al anterior, quitando la columna de las etiquetas y las unidades de los datos como los 'kmpl' o 'bhp'. La única diferencia es que meto los datos de entrenamiento y test en el mismo *DataFrame*, para así no tener que repetir el preprocesado.

Los cambios que vienen a continuación estarán en el notebook '*guion-final.ipynb*'. Lo primero que hacemos es ver la cantidad de variables categóricas que tenemos y decidir cuales vamos a utilizar. En principio, sólo el nombre o marca del vehículo es la que más categorías únicas tiene, por lo que es la que nos planteamos si quitar o mantener (a priori la quitaremos). Lo siguiente que vemos es que la categoría

'Combustible' predominan las de tipo *Diesel* y *Petrol*, por lo que el resto que no sean de ese tipo, las clasificaremos como *Other*.

A continuación, vamos a transformar estas variables categóricas en variables 'dummy', es decir, variables numéricas mediante una serie de ceros y unos.

Ciudad_L	Combustible_Diesel	Combustible_Other	Combustible_Petrol	Tipo_marchas_Automatic	Tipo_marchas_Manual	Mano_First	Mano_Fourth & Above	Mano_Second	Mano_Third
0	1	0	0	0	1	1	0	0	0
0	0	0	1	0	1	1	0	0	0
0	1	0	0	0	1	1	0	0	0
0	0	0	1	0	1	1	0	0	0
0	0	0	1	0	1	1	0	0	0
0	1	0	0	0	1	0	0	1	0
0	1	0	0	0	1	1	0	0	0
0	0	0	1	1	0	1	0	0	0
0	0	0	1	0	1	1	0	0	0
1	1	0	0	0	1	1	0	0	0

Ahora pasamos a tratar los datos incorrectos o que faltan. Esta vez, lo vamos a hacer de una forma distinta a la anterior. Vamos a rellenar los datos que faltan en una columna, utilizando la mediana de esos datos en esa misma columna. Con el descuento, al igual que hicimos anteriormente, pienso que es mejor completarlos con un 0.

Algo nuevo que haremos será tratar los valores con ruido o *outliers*. Para ello vamos a utilizar la siguiente función:

```

1  def find_outliers(x):
2      q1 = np.percentile(x, 25)
3      q3 = np.percentile(x, 75)
4      iqr = q3-q1
5      floor = q1 - 1.75*iqr
6      ceiling = q3 + 1.75*iqr
7      outlier_indices = list(x.index[(x < floor) | (x > ceiling)])
8      outlier_values = list(x[outlier_indices])
9
10     return outlier_indices, outlier_values

```

En ella obtenemos un valor para los cuartiles 1 y 3 (podemos ajustarlo manualmente para hacerlo más o menos restrictivo), e imprimimos los valores que queden fuera de este rango. Gracias a esto, podemos observar que, en los kilómetros por ejemplo, hay un coche con 6 millones de kilómetros realizados, lo cual es prácticamente imposible. Se ha seleccionado un valor umbral de medio millón de kilómetros realizados para que selecciones estos valores como ruido (con más de 300.000 suelen ir a desguace en vez de venderse). También se han ajustado otros valores como el consumo, en el algunos coches tenían un 0.

Lo siguiente que se ha realizado ha sido añadir interacciones entre características. Esto puede resultar muy útil, ya que añadimos más características a nuestro

modelo que pueden potenciar el resultado si los atributos generan alguna relación. Hay que tener en cuenta que las interacciones entre variables que pertenecen a la misma variable categórica, son siempre cero. Para llevarlo a cabo, se ha utilizado *'PolynomialFeatures'*, una función de *sklearn* cuyo funcionamiento será más fácil de comprender si se mira en la página oficial [10].

Tras realizar todo este preprocesado, hemos obtenido una cantidad de 970 columnas en nuestro dataset. Esto nos lleva a la siguiente parte del código, que es la reducción de dimensionalidad mediante PCA o selección de características (*SelectKBest*). Esto finalmente no se va a utilizar, ya que estaba pensado para hacer algún intento más utilizando redes neuronales, y no ha sido posible.

Las pruebas que se han realizado con estos datos son bastante más de las que se han llegado a subir (las añadire todas al zip), ya que las subidas diarias limitaban el poder probar cada cambio. Pese a esto, no creo que fuesen mucho mejores de las que ya hay, ya que en la prueba rápida realizada, los valores se mantenían más o menos similares.

Para empezar se probó el último modelo visto, *StackingClassifier*, con la misma configuración, para así tener una referencia de si estaba funcionando o no. El resultado fue bastante bueno, ya que obtuve un 0.77825, colocándome el 18 en la clasificación. Esto quería decir que los cambios en los datos habían funcionado.

En la tabla del siguiente punto explicaré los detalles entre unas subidas y otra, pero los cambios o pruebas más relevante que hice, me gustaría comentarlas ahora. Uno de ellos fue la búsqueda automática de hiperparámetros para nuestro modelo. Esto lo he hecho gracias a *GridSearchCV*, una función a la que le introducimos una serie de hiperparámetros para un modelo y, mediante validación cruzada, selecciona los que mejores resultados han ofrecido:

```
1  # Creacion del grid de parametros
2  param_grid = {
3      'criterion': ['gini'],
4      'max_depth': [None],
5      'min_samples_leaf': [1, 2, 3, 4, 5],
6      'min_samples_split': [2, 3, 4, 5, 6],
7      'n_estimators': [100, 150, 175, 200]
8  }
9
10 # Entrenamos el modelo con las distintas opciones
11 gs = GridSearchCV(estimator=rf, param_grid=param_grid,
12                  cv=5, n_jobs=-1, verbose=1)
13 gs = gs.fit(x_train, y_train)
14
15 # Imprimimos los mejores resultados
16 print(gs.best_score_)
17 print(gs.best_params_)
18
```

```

19 # Creamos el modelos con los parametros elegidos
20 bp = gs.best_params_
21 RndForestClf = RandomForestClassifier( criterion=bp[ 'criterion' ],
22                                     min_samples_leaf=bp[ 'min_samples_leaf' ],
23                                     min_samples_split=bp[ 'min_samples_split' ],
24                                     max_depth=bp[ 'max_depth' ],
25                                     n_estimators=bp[ 'n_estimators' ])

```

Este ejemplo calcula unos parametros para el *RandomForest*, con el cual realizaría tres subidas (aunque tienen algunos cambios en el preprocesado que explicare en la tabla) que me dieron un resultado de 0.76617, 0.76186 y 0.76531. Valores muy similares al anterior pero con los que no conseguí mejorar.

La siguiente subida sería con la que he alcanzado mi posición final. Lo que he hecho ha sido utilizar el *GridSearchCV* para generar un modelo de ***Random-Forest***, y a su vez, meterlo dentro de un ***StackingClassifier*** junto con una ***Linear Support Vector Machine***. La idea la he sacado del propio manual del *StackingClassifier* [3]. Sinceramente, todavía me esperaba un mejor resultado, ya que en las pruebas rápida realizada he alcanzado un valor de hasta 82-85 % de precisión, y según la tendencia de estos experimentos rápidos, este valor solía ser menor que el obtenido finalmente en la web. Aun así, el *accuracy* obtenido ha sido de 0.79119, con el que conseguí la posición 12 si no recuerdo mal.

Otro cambio bastante sustancial para las últimas ejecuciones fue el de introducir aún más modelos en el *StackingClassifier*:

```

1 estimators = [
2     ( 'lr' , LogisticRegression() ),
3     ( 'RndForestClf' , RndForestClf ),
4     ( 'svr' , make_pipeline( StandardScaler() , LinearSVC() ) ),
5     ( 'bayes' , GaussianNB() )
6 ]
7
8 StackingClf = StackingClassifier( estimators=estimators ,
9                                 final_estimator=LogisticRegression() )

```

y también una última versión con un *ExtraTreeClassifier* con una búsqueda de parámetros antes, pero no mejoró el resultado.

Finalmente, comentar que me habría gustado probar bastantes más opciones y modelos, como pueden ser las redes neuronales con este último preprocesado, probar con la reducción de características o investigar si encuentro alguna forma de conseguir mejorar el que ya tengo. No obstante, estoy bastante contento con el resultado final, ya que no he tenido mucho tiempo y siento que el resultado ha sido bastante competitivo (está a un 4 % de precisión aproximadamente del mejor).

3. Tabla de soluciones

Las palabras subrayadas te llevarán a la explicación completa en la memoria. Los valores con un '-' es porque al no mejorar, la posición sería la misma o más baja, ya que si mejoraba algún otro alumno, cambiaría. (posición/total de alumnos).

Fecha	Posición	Train/Test score	Descripción preprocesado	Descripción método	Parámetros
Dec 26 - 03:34	20/22	0.685602 0.61518	Preprocesado de prueba explicado en la sección 2.1	Red neuronal en forma de <u>MLPerceptron</u>	hidden_layer_sizes = (100,100)
Dec 26 - 04:00	16/22	0.671908 0.72131	Preprocesado de prueba explicado en la sección 2.1	<u>Random Forest</u> por defecto	n_estimators = 200
Dec 27 - 02:56	-	0.7462 0.46764	Preprocesado de prueba explicado en la sección 2.1	<u>Red neuronal</u> hecha a mano	Capas dense de (12-10-10-6) optimizer = Adam(0.0001) activation = 'sigmoid'
Dec 27 - 03:04	-	0.69864 0.69111	Preprocesado de prueba explicado en la sección 2.1	<u>Random Forest</u> por defecto	n_estimators = 100 max_depth = None
Dec 28 - 22:30	-	0.71232 0.71182	Preprocesado de prueba con pequeñas <u>modificaciones</u>	<u>Random Forest</u> por defecto	n_estimators = 100 max_depth = None (obtuvieron mejor resultado en el train que el anterior)
Dec 28 - 22:34	20/30	0.71156 0.75323	Preprocesado de prueba con pequeñas <u>modificaciones</u>	<u>Random Forest</u> por defecto	n_estimators = 100 max_depth = None (obtuvieron mejor resultado en el train que el anterior)
Dec 28 - 22:39	-	0.72455 0.75150	Preprocesado de prueba con pequeñas <u>modificaciones</u>	<u>Random Forest</u> por defecto	n_estimators = 100 max_depth = None (obtuvieron mejor resultado en el train que el anterior)
Dec 29 - 01:17	-	0.72463 0.74719	Preprocesado de prueba con pequeñas <u>modificaciones</u>	Stacking Classifier con un <u>Random Forest</u> y <u>LinearSVC</u>	final_estimator = LogisticRegression()
Dec 29 - 01:33	-	0.72003 0.75237	Preprocesado de prueba con pequeñas <u>modificaciones</u>	Stacking Classifier con un <u>Random Forest</u> y <u>LinearSVC</u>	final_estimator = LogisticRegression(max_iter = 10000, tol = 1e-5)
Dec 29 - 02:47	-	0.69329 0.63589	Preprocesado de prueba con pequeñas <u>modificaciones</u>	Stacking Classifier con un <u>AdaBoostClassifier</u> , <u>SVC</u> y <u>SGDClassifier</u>	final_estimator = LogisticRegression()
Dec 30 - 01:00	-	0.72387 0.70750	Preprocesado de prueba con pequeñas <u>modificaciones</u> Añadimos nombre y marca del coche	Stacking Classifier con un <u>Random Forest</u> y <u>LinearSVC</u>	final_estimator = LogisticRegression()
Dec 30 - 01:33	-	0.72546 0.71786	Preprocesado de prueba con pequeñas <u>modificaciones</u> Utilizamos <u>MinMaxScaler</u> para normalizar los datos	Stacking Classifier con un <u>Random Forest</u> y <u>LinearSVC</u>	final_estimator = LogisticRegression()
Dec 30 - 19:15	18/40	0.81402 0.77825	Nuevo preprocesado más complejo y con bastantes detalle resumidos <u>aquí</u>	<u>Stacking Classifier</u> con un <u>Random Forest</u> y <u>LinearSVC</u>	final_estimator = LogisticRegression()
Dec 31 - 01:02	-	0.80722 0.76617	Nuevo preprocesado más complejo y con bastantes detalle resumidos <u>aquí</u> Se procesan los datos con ruido	<u>Random Forest</u> por defecto	Búsqueda de parámetros con <u>GridSearchCV</u>
Dec 31 - 01:03	-	0.81531 0.76186	Nuevo preprocesado más complejo y con bastantes detalle resumidos <u>aquí</u> Se procesan los datos con ruido	<u>Random Forest</u> por defecto	Búsqueda de parámetros con <u>GridSearchCV</u> El criterion = 'entropy', que fue lo más destacable
Dec 31 - 01:24	-	0.81531 0.76531	Nuevo preprocesado más complejo y con bastantes detalle resumidos <u>aquí</u> Se procesa el ruido Añadimos la columna nombre	<u>Random Forest</u> por defecto	Búsqueda de parámetros con <u>GridSearchCV</u>
Jan 1 - 18:09	12/45	0.82998 0.79119	Nuevo preprocesado más complejo y con bastantes detalle resumidos <u>aquí</u> Se procesa el ruido Añadimos la columna nombre	Stacking Classifier con un <u>Random Forest</u> y <u>LinearSVC</u>	Búsqueda de parámetros del <u>RandomForest</u> utilizado con <u>GridSearchCV</u>
Jan 1 - 22:06	-	0.82444 0.78602	Nuevo preprocesado más complejo y con bastantes detalle resumidos <u>aquí</u> Se procesa el ruido Añadimos la columna nombre	Stacking Classifier con un <u>LogisticRegression</u> , <u>Random Forest</u> , <u>LinearSVC</u> y <u>GaussianNB</u>	Búsqueda de parámetros del <u>RandomForest</u> utilizado con <u>GridSearchCV</u>
Jan 1 - 22:37	-	0.82238 0.78774	Nuevo preprocesado más complejo y con bastantes detalle resumidos <u>aquí</u> Se procesa el ruido Añadimos la columna nombre	Stacking Classifier con un <u>LogisticRegression</u> , <u>Random Forest</u> , <u>ExtraTreeClassifier</u> , <u>LinearSVC</u> y <u>GaussianNB</u>	Búsqueda de parámetros del <u>RandomForest</u> utilizado con <u>GridSearchCV</u>

Referencias

- [1] Scikit-Learn *SVC*
<https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC>
- [2] Scikit-Learn *RandomForestClassifier*
<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier>
- [3] Scikit-Learn *StackingClassifier*
<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingClassifier.html>
- [4] Scikit-Learn *MLPClassifier*
https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier
- [5] Scikit-Learn *plot_learning_curve*
https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html#sphx-glr-auto-examples-model-selection-plot-learning-curve-p
- [6] Isaac Changhau. *Loss Functions in Neural Networks*
https://isaacchanghau.github.io/post/loss_functions/
- [7] DePaul University.
<http://mdp.cdm.depaul.edu/DePy2016>
- [8] Scikit-Learn. *StandardScaler*
<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>
- [9] Scikit-Learn *PCA*
<https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html>
- [10] Scikit-Learn *PolynomialFeatures*
<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html>
- [11] Scikit-Learn *GridSearchCV*
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
- [12] MathWorks *Support Vector Machine*
<https://es.mathworks.com/discovery/support-vector-machine.html>

- [13] Scikit-Learn *cross_val_score*
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html
- [14] Scikit-Learn *train_test_split*
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
- [15] Scikit-Learn *Confusion Matrix*
https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html