Datathon Management Solutions

Manuel Jesús Galán Moreu

Pablo Lázaro Herrasti

19 / 01 / 2020

Contents

Introducción y objetivo

Automated machine learning (AutoML) es el proceso de automatización de un proceso de machine learning a problemas del mundo real. AutoML se encarga desde el preprocesamiento con el tratado de los datos hasta de realizar el modelo que entrene dichos datos.

El objetivo de este Datathon es diseñar un modelo de Auto ML que sea capaz de adaptarse para realizar buenas predicciones en diferentes datasets sin realizar modificaciones en el código.

Bibliotecas

Para los diferentes modelos y técnicas de Machine Learning que se utilizarán durante el reto se ha hecho uso de las librerías y bibliotecas que hoy en día destacan en el mercado que facilitan la programación y análisis de los datos, así como el modelado de estos para obtener la solución al reto. Entre otras, **pandas** y **scikit-learn**, las cuales, se explicarán brevemente a continuación, además de **hyperopt** para el bayesian optimization de hiperparámetros. Además, para tener el código lo más estandarizado posible siguiendo uno de los requisitos del reto, se ha hecho uso de la librería flake8, aparte de la lectura de la documentación pep8. Con ambas, se ha llegado a la estandarización de algunos aspectos, entre otros, los que se exponen a continuación:

- No sobrepasar los 79 caracteres por línea.
- Creación de los métodos privados excepto auto_ml que es de uso público.
- Comentarios en todos los métodos y dentro de cada método su estructura a seguir.
- Sintaxis adecuada: espacio detrás de delimitador coma (,).
- Nombre de las funciones en minúscula.

Pandas y scikit-learn

Pandas es una biblioteca de código abierto con licencia BSD que proporciona estructura de datos y herramientas de análisis de datos de alto rendimiento y fáciles de usar. **Python** ha sido siempre excelente para la mezcla y preparación de datos, pero no tanto para el análisis y modelados de estos. Por tanto, el uso de esta biblioteca sirve para llenar este vacío sin tener que cambiar a otro lenguaje como podría ser R.

Cabe destacar que pandas no implementa una funcionalidad de modelado significativa fuera de la regresión lineal. Es por esto, que la segunda gran librería de la que se hace uso para llevar a cabo el reto es scikit-learn. Scikit-learn o sklearn como algunos lo llaman, es una biblioteca para Machine Learning de software libre usada para el lenguaje de programación Python. En esta se incluyen varios algoritmos de clasificación, regresión entre otros como pueden ser **Support Vector Machine, Gradient Boosting o K-means**, los cuales se explicarán en apartados posteriores más detalladamente.

Procesamiento general

A modo de introducción, en el presente capítulo, se intenta reflejar los diferentes pasos que se han llevado a cabo para hacer frente al reto de AutoML.

Como cualquier reto de machine learning, la principal tarea y la más importante es la del **tratamiento de los datos**. Para ello se han aplicado algunas técnicas muy conocidas ya hoy en día como son la eliminación de duplicados o de información no útil, transformación a binario de las features o aplicar la correlación para identificar la relación que puede existir entre las variables, y entre las features, y la variable a predecir. Por otro lado, se han automatizado algunas otras funciones, bien para que se obtenga como resultado el AUC, que es la métrica requerida como solución al reto, o para poder leer el dataframe que se quiera introducir, entre otras.

Como segundo paso, y no por ello menos importante, se realiza el modelo con el que se entrenarían los datos para el posterior testeo o prueba. Se ha hecho uso de diferentes modelos, aplicando desde árboles de decisión como Random Forest, o desde clasificación binaria como SVM (Support Vector Machine), así como de diferentes técnicas que permiten buscar los mejores hiperparámetros que conducen a la solución con resultados más precisos. Para poder aplicar dichas técnicas como Grid Search o la optimización bayesiana, se ha inicializado diferentes parámetros con valores entre los que pueden encontrarse la mejor solución, para no tener que realizar este proceso de forma manual y tediosa.

Tal y como se ha planteado el problema, tras el análisis de datos y pruebas con los diferentes modelos, se ha considerado que la mejor solución consiste en una función que ejecute un modelo o modelos dependiendo del que se adapte o se adapten mejor a los datos de entrada. De esta manera, la función principal realiza una primera aproximación con los modelos que se han considerado que son los más eficaces. Una vez realizada esta, se realizará un ordenamiento de mejor a peor en cuanto a resultado. Con estos ordenados, solamente los datos serán entrenados con aquellos modelos, cuyo valor en la aproximación primera, haya variado un valor de 0,15 con respecto el mejor resultado (tras probar con distintos datos). Además, no conformes con encontrar el mejor resultado de esta manera, una vez finalizado este proceso, la función ejecuta la técnica de Voting Ensemble con los tres mejores modelos. Con el resultado del primer proceso y del Voting, se comparan ambos y se obtiene el mejor valor de AUC. Con la técnica de Voting podemos entrenar varios modelos de aprendizaje automático con los mismos datos. Cuando tengamos datos nuevos, obtendremos una predicción de cada modelo. Cada modelo tendrá asociado un voto. De esta forma, propondremos como predicción final lo que voten la mayoría de los modelos.

Con respecto a los modelos considerados necesarios a aplicar para el presente reto no han sido otros que **Support Vector Machine o SVM, Random Forest, XGBoosting, Logistic Regression, KNearestNeighbors, Naive Bayes y LigthGBM** como modelo algo innovador a los anteriores ya tan conocidos. La decisión de elegir estos y no otros no ha tenido nada más que ver con el estudio y análisis de los datos, así como de los datasets que se han presentado, el número de features y de samples que se nos han proporcionado.

En primer lugar, los datasets que se han facilitado no son lo suficientemente grandes como para hacer una Neural Network o una Multilayer Perceptron que tenga que aprendar tal cantidad considerable de datos aplicando Deep learning. Creemos que el Deep learning es sobretodo aplicable a grandes cantidades de datos, ya que si no se puede llegar al overfitting muy fácilmente. Por otro lado, aunque no se demuestre en el presente documento, se han realizado pruebas de Principal Component Analysis (PCA) con la que no se mejoraron los resultados obtenidos, ni aplicando algunas técnicas de ensemble como Bagging y Stacking. La otra opción que se podría tener en cuenta a la hora de realizar un modelo es una algún tipo de

regresión. En este caso, se han hecho pruebas con la regresión logística, dando valores menos favorables en cuanto a la solución que se desea llegar, aunque se han tenido en cuenta por si como input de entrada se introdujesen valores que se ajustasen mejor a este tipo de modelo (datos con features más simples). Además, no probando por suerte con todos sino aplicando también el conocimiento, se puede observar que la feature "*Class*", que es la que se intenta predecir y la que nos define si el modelo es bueno o no, es de tipo binario. Esto significa que tomar decisiones con algoritmos como SVM (binario) puede ser más que una buena idea. Como bien se sabe también, la idea de aplicar árboles de decisión es siempre un acierto y más cuando se trata de un problema con clasificación binaria. Los árboles de decisión van generando sus ramas siguiendo un enfoque de división binaria recursiva. Por esta misma razón, se ha investigado algo más acerca de otros algoritmos basados en árboles, utilizando uno que hoy en día es algo menos conocido como Light GBM. Light GBM es un framework de tipo gradient boosting que usa árboles como algoritmo de aprendizaje. Además, este algoritmo desarrolla las ramas de sus árboles de manera vertical mientras que la mayoría crece horizontalmente, como si le creciesen más hojas dentro de la misma rama del árbol. Esta manera de crecer provoca que se escoja la hoja con pérdida máxima delta para crecer. Cuando se vuelve a crecer, el algoritmo puede reducir más pérdidas que un algoritmo de nivel inteligente. Tras probar dicho algoritmo, nos encontramos con el mismo caso que la red neuronal. Este es muy potente y rápido, pero está considerado para dataset de un tamaño bastante considerable, por lo que, se ha tenido en cuenta solo por si el dataset de entrada que se quiera analizar fuese lo suficientemente grande ya que sino provocaría overfitting en nuestro modelo.

class DatathonML:

En el epígrafe de la clase Class DatathonML se va a explicar en detalle las principales funciones que se han utilizado en la clase para que cualquier persona pueda hacer uso de él en caso de necesitarlo o mejorarlo. Para facilitar la comprensión, se va seguir el orden descendente en el que se encuentran las funciones del código. Además, se hace referencia al apartado de Anexos dónde se encuentran cada uno de los métodos implementados con comentarios explicativos

1. ***def __init__().*** Inicialización de la clase en dónde se inicializan los hiperparámetros y parámetros de los modelos Naive Bayes (NB), Support Vector Machine (SVM), Random Forest (RF), XGBoost, Logistic Regression (LR) y LGBLight (LGB). Además, se declara ***self.all_possible_models*** para la función que va seleccionando los modelos que se ejecutarán, así como ***self.results*** para guardar los resultados. Para más detalle del código, puede hacer referencia al .Anexo 1: def __init__()

2. ***def r**ead_dataframe().* Función que devuelve el dataframe que se introduzca. (Anexo 2: def read_dataframe())

3. ***def obtain_labels().***

   Teniendo en cuenta que el dataset data_yeast1 tiene como **Class negative o false**, se ha considerado oportuno realizar una función que **transforme a binario dicha feature o columna**. De esta manera, se asegura que cualquier columna pasada como etiqueta es fácilmente transformable. Anexo 3: def obtain_labels()

4. ***def preprocess_dataframe().*** (Anexo 4: def preprocess_dataframe())

   Por otro lado, con la finalidad de eliminar aquella información que no es relevante para nuestro modelo, se han eliminado las filas duplicadas y aquellas columnas que no proporcionan información útil, es decir, aquellas features o columnas que no cambian de valor para ninguna fila. Además, al leer el dataset, las columnas son de tipo object y no se reconocen valores numéricos en el dataframe. Por lo tanto, se aplica una transformación en la que se reemplazan las comas por puntos para poder pasar todas las celdas a numérico.

5. ***Función de correlacion*** (Anexo 5: Correlación)

   Se ha considerado oportuno realizar la correlación entre las diferentes variables. Se puede decir que dos variables están correlacionadas cuando existe una relación entre sus valores, si aumenta el valor de una variable la otra también aumenta o disminuye. En general, el mejor método para buscar correlaciones entre las distintas variables del dataset es crear una matriz de correlación (utilizando por ejemplo la correlación de Pearson). Una vez obtenida esta matriz, eliminamos aquellas variables que tengan una correlación con otra variable mayor que un threshold (en nuestro caso, este threshold será igual a 0.8)

6. ***def compute_AUC().*** Función que permite calcular la métrica AUC. (Anexo 7: def compute_AUC ()Anexo 7: def compute_AUC ())

7. ***def training_function_bayes().*** (Anexo 8: def training_function_bayes ())

   Función de training para el caso que se aplique el algoritmo clasificador Light GBM. En este caso, la optimización será por Bayes, aplicándose los hiperparámetros que se comentaban en la función de inicialización y una cross-validation. Cross-validation (cv) consiste en un modelo de técnicas para evaluar como los resultados de un análisis estadístico se generalizarán a un conjunto de datos independiente. Se utiliza principalmente en entornos donde el objetivo es la predicción, y uno desea estimar con qué precisión funcionará un modelo predictivo en la práctica. En nuestro caso, se ha optado por utilizar un valor de 5 para la división del dataset en cada entrenamiento.

8. ***def select_best_models().*** (Anexo 9: def select_best_models ())

   La presente función, como primer objetivo, tiene el de ordenar descendentemente los modelos en función de resultados más óptimos a menos. El segundo objetivo, se divide en dos caminos. Si el valor de flag se encuentra a 0, los modelos serán ordenados, y siempre que tenga una diferencia menor de 0.15 en cuanto al resultado, se adjuntará a una lista. En caso de que la diferencia con respecto el que está una posición por encima supere los 0.15, el modelo no será introducido en la lista y no se evaluará. El flag se encontrará a 0 después de haber realizado una primera aproximación con valores por defecto de todos los modelos que se han tenido en cuenta para el reto.

   Si el valor del flag se encuentra a 1, simplemente se compararán todos los modelos ordenados y se creará una tupla con aquel que haya obtenido un mayor resultado de AUC. El valor del flag se encuentra a 1 cuando, tras la primera aproximación, se descartan los modelos que obtuvieron menor valor de AUC, y vuelven a ser entrenados teniendo en cuenta una optimización de GridSearch o por Bayes.

   Por último, si el valor del flag se encuentra a 2, esto significa que se va a aplicar un Ensemble de voto con los 3 mejores modelos, por lo que, los modelos serán ordenados de mejor a peor y se seleccionarán solo los 3 mejores para aplicarse dicha técnica.

9. ***def first_approximation_training().*** (Anexo 10: def first_approximation_training ())

   Primera aproximación de los modelos para poder comprobar entre qué valores aproximadamente se encontrarán los valores de AUC de salida de los modelos tras realizar optimizaciones y búsqueda de hiperparámetros. Aquí, los valores de os hiperparámetros que se introducen son por defecto.

10. ***def training_function_grid().*** (Anexo 11: def training_function_grid ())

    Función idéntica que la de optimización por Bayes, pero en este caso, se realiza un Grid Search para la búsqueda de aquellos valores que obtengan los mejores resultados. Esta función se aplica al resto de modelos que no es Light GBM.

11. ***def __ensemble_voting()*** (Anexo 12: def __ensemble_voting ())

    Función que escogiendo los tres mejores modelos tras la primera aproximación crea un modelo de Ensemble en el que se decide la solución por sistema de voto por mayoría.

12. ***def __print_final_solution()*** (Anexo 13: def __print_final_solution ())

    Función que imprime los resultados en cuanto al modelo, AUC y parámetros de Grid Search.

13. def auto_ml(self, df). (Anexo 14: def auto_ml ())

Función principal que se encarga de llamar al resto de funciones para la limpieza de los datos, además de, realizar el 'split' tanto de entrenamiento como de prueba o testeo y realizar el entrenamiento para poder extraer el mejor valor de AUC final.

Como usar la clase

Para poder utilizar dicha clase, y con la finalidad de que sea lo más sencillo posible, se ha subido todo el código junto con el main principal a un github en la url: https://github.com/polazaro/DatathonMS

Los pasos a seguir para poder poner en marcha el código junto con los requerimientos para que pueda funcionar correctamente son los que se describen a continuación.

Requerimientos:

1) Dataset de entrada en formato csv.
2) Instalación de Spyder y Python 3.
3) Bibliotecas de sklearn, pandas, xgboost, lightgbm e hyperopt que se describen en el import del código.

Pasos:

1) Importar la clase de la siguiente manera:

```
from datathon_ml import DatathonML
```

2) Crear el objeto de la clase que contiene la función auto_ml:

```
datathon_ml = DatathonML()
```

3) Leer el dataframe en caso que sea necesario con la dirección correspondiente:

```
dir_dataset = 'Ruta PATH de la carpeta'
```

4) Llamar al método de la clase auto_ml y pasar por parámetros el dataframe leído previamente:

```
final_results = datathon_ml.auto_ml(df)
```

Caso de uso

En el caso de uso se explica cómo se ha afrontado el problema de primeras con los datos y datasets que se nos han proporcionado, que han ayudado a la solución planteada, obteniendo resultados válidos con los modelos que anteriormente se explicaban. Para ello, se ha hecho un proceso de análisis de los datos, un procesamiento de ellos y el modelo o modelos para su aprendizaje. Se tiene en cuenta que, en el presente caso de uso, la distribución de datos de training y datos de testing se encuentran partidos en un 80/20, división que saca resultados algo peores que la que se ha tenido en cuenta en la función de AutoML, que es 70/30.

Análisis de datos

Los primeros pasos que se han seguido para llevar a cabo el caso de uso son la lectura de los datos facilitados en formato 'csv' y mostrar los mismos. Estos pasos se han realizado gracias a la función '**read_csv**' y '**head**()' que pandas nos facilita y con la que se puede leer el **dataframe** creado previamente y mostrar luego las *n* primeras filas respectivamente.

De lo anteriormente comentado se han podido sacar algunos datos que pueden servir de interés, mostradas en la siguiente tabla.

Dataset	Dimensión	Tipo de datos	Class
data_elephants	1391x231	Numéricos	0/1
data_rings	7400x21	Numéricos	0/1
data_yeast1	1484x9	Numericos	Positive/Negative

Tabla 1. Propiedades DataSets

En la siguiente imagen se puede observar la salida de los datos tras aplicar head(). La Ilustración 1 hace referencia al dataset de data_elephants, la Ilustración 2 a data_rings y la Ilustración 3 a data_yeast1.

	Atr-1	Atr-2	Atr-3	Atr-4	Atr-5	Atr-6	Atr-7	Atr-8	Atr-9	Atr-10	...	Atr-222	Atr-223	Atr-224	Atr-225	Atr-226
0	-0,28698	-0,624297	-0,679333	-0,455715	0,371213	-0,415471	-0,549112	-0,4722	0,363383	0,33419	...	0	-0,049855	120.049	-0,078862	-0,021452
1	0,363763	0,146879	-0,83422	0,819986	0,504474	167.231	-0,774634	0,76555	0,355781	0,215197	...	0	-0,049855	129.994	-0,078862	-0,021452
2	-0,15592	-0,139025	-115.844	246.346	-0,463596	-138.827	-0,945256	260.892	-0,049767	0,846782	...	0	-0,049855	0,031608	0,182763	-0,021452
3	-122.972	-119.093	-0,364069	-132.571	-0,005645	-0,740226	-0,468026	-131.328	-0,082819	0,857917	...	0	-0,049855	0,591019	-0,078862	-0,021452
4	0,9162	0,265188	-0,252377	256.115	-0,128087	-0,105623	-0,302654	259.287	-0,167432	-0,626084	...	0	-0,049855	-0,114025	0,167061	-0,021452

5 rows × 231 columns

Ilustración 1. Data_elephants

	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	...	A12	A13	A14	A15	A16	A17	A18	A19	A20	Class
0	849	2177	598	1689	3114	-3406	3799	-2642	1578	-181	...	1633	4664	1081	-1172	-166	578	1664	309	-3028	0
1	947	543	782	-449	-8	1316	756	-61	1604	1686	...	-382	1259	608	-2292	1008	2687	-1422	-606	2374	1
2	759	-314	898	-1743	2003	1117	911	136	-489	-144	...	112	598	474	2026	1431	-336	79	1123	302	1
3	531	1374	517	1226	1122	-4	-1227	2277	1083	888	...	1063	2454	587	-744	1216	916	-476	6	-477	1
4	-1443	1065	2071	207	-865	-786	2655	-366	971	-341	...	4195	-117	-2694	14	4097	1356	-944	-602	2348	0

5 rows × 21 columns

Ilustración 2. Data_rings

	Mcg	Gvh	Alm	Mit	Erl	Pox	Vac	Nuc	Class
0	0,58	0,61	0,47	0,13	0,5	0	0,48	0,22	negative
1	0,43	0,67	0,48	0,27	0,5	0	0,53	0,22	negative
2	0,64	0,62	0,49	0,15	0,5	0	0,53	0,22	negative
3	0,58	0,44	0,57	0,13	0,5	0	0,54	0,22	positive
4	0,42	0,44	0,48	0,54	0,5	0	0,48	0,22	negative

Ilustración 3. Data_yeast1

Procesamiento

Una vez que se han analizado los datos, es importante hacer la limpieza de los mismos para que los modelos de aprendizaje automático puedan predecir tras la introducción de nuevos datos en un futuro. Para ello, se han utilizado las funciones ya explicadas en el capítulo anterior. Los pasos que se han seguido son:

1) **Transformación a binario de las features**
2) **Eliminar duplicados e información no útil.**

Con los pasos 1 y 2 realizados, se vuelve a hacer un proceso de mostrar dichos datos para comprobar si el preprocesamiento realizado ha tenido relevancia y si hemos conseguido eliminar información no útil.

La tabla anterior (Tabla 1), por tanto, ha sufrido algunos cambios tal y como se muestra a continuación en la Tabla 2.

Dataset	Dimensión	Tipo de datos	Class
data_elephants	1391x110	Numéricos	0/1
data_rings	7400x20	Numéricos	0/1
data_yeast1	1453x8	Numericos	0/1

Tabla 2. Datasets de entrada

Se puede comprobar como el número de filas y columnas ha sido reducido en alguno de los datasets.

3) **Correlación entre features**

El penúltimo paso ha sido realizar la correlación entre las variables para ver la relación entre sus valores y poder eliminar variables muy correlacionadas, es decir, que nos estén dando información similar al resultado final. La correlación utilizada es la de Pearson, ya que es la que mejor funciona en estos casos por su naturaleza positiva y negativa.

4) **Correlación entre features y target**

El último paso consiste en realizar la correlación entre features y target, para ver cuales pueden tener más importancia y cuales menos. Finalmente, nos quedamos con el 75% de features que más correladas están con el target, para evitar bajadas de accuracy.

Modelos

En el presente epígrafe se exponen los diferentes modelos que se han utilizado, así como los resultados obtenidos.

Antes de entrenar y testear con los datos que se nos facilitan, se ha aplicado algunas técnicas que permitirán que el modelo pueda conseguir obtener los mejores resultados. Para ello se aplica **GridSearch** y cross-**validation** entre otros. Además, se realiza un pipeline con el modelo que se usa junto a un tipo de estandarización. La estandarización de un conjunto de datos es un requisito común para muchos estimadores de Machine Learning. En este caso se han probado StandarScaler y MinMaxScaler. Como en todos los datasets los datos están en general en dimensiones parecidas, se ha optado por utilizar **StandarScaler**. Por otro lado, se ha aplicado aquí el feature selection mirando la correlación entre feature y label con **SelectPercentile** (75%) de sklearn.

Con todo lo anteriormente comentado, se consiguió mejorar los resultados obtenido en la primera predicción ya que como se sabe localiza los valores óptimos, por lo que a partir del primer modelo ya se decidió utilizarse en el resto.

1) Support Vector Machine

GridSearch

```
parameters = {'SVM__C':[0.001,0.1,10,100,10e5], 'SVM__gamma':[0.1,0.01]}
grid = GridSearchCV(pipeline, param_grid=parameters, cv=5)
```

Resultados

Dataset	Accuracy	AUC	Hiperparámetro 1	Hiperparámetro 2
data_elephants	0.7527	0.7514	SVM__C:10	SVM__gamma:0.01
data_rings	0.9811	0.9811	SVM__C: 0.1	SVM__gamma:0.1
data_yeast1	0.8179	0.7055	SVM__C:100	SVM__gamma:0.01

Tabla 3. Resultados SVM

2) Random Forest

GridSearch

```
parameters = {'clf__n_estimators':[100, 300, 500, 800, 1200],
'clf__max_depth':[5, 8, 15, 25, 30],
'clf__min_samples_split':[2, 5, 10, 15, 100],
'clf__min_samples_leaf':[1, 2, 5, 10]
}

grid = GridSearchCV(pipeline, param_grid=parameters, cv=5)
```

Resultados

Dataset	Accuracy	AUC	Hiperparámetro 1	Hiperparámetro 2
data_elephants	0.846	0.845	clf__max_depth: 25	clf__n_estimators: 500
data_rings	0.957	0.957	clf__max_depth: 25	clf__n_estimators: 1200
data_yeast1	0.808	0.705	clf__max_depth: 25	clf__n_estimators: 100

Tabla 4. Resultados Random Forest

3) XG Boost

GridSearch

```
parameters = {'xgbc__n_estimators': range(60, 220, 40),
'xgbc__max_depth':[3, 4, 5],
}

grid_xgbc = GridSearchCV(pipeline, param_grid=parameters, cv=5)
```

Resultados

Dataset	Accuracy	AUC	Hiperparámetro 1	Hiperparámetro 2
data_elephants	0.8208	0.8233	xgbc__max_depth: 4	xgbc__n_estimators: 180
data_rings	0.9635	0.9633	xgbc__max_depth: 3	xgbc__n_estimators: 180
data_yeast1	0.8110	0.7248	xgbc__max_depth: 3	xgbc__n_estimators: 60

Tabla 5. Resultados XG Boost

4) Naive Bayes

```
nbc = GaussianNB()
nbc.fit(X_train1, y_train1)
print("Elephant accuracy NB = %3.4f" %(nbc.score(X_test1,y_test1)))
fpr, tpr, thresholds = metrics.roc_curve(y_test1, nbc.predict(X_test1))
print("Elephant AUC NB = %3.4f" %(metrics.auc(fpr, tpr)))
```

Dataset	Accuracy	AUC
data_elephants	0.9653	0.7180
data_rings	0.9811	0.9811
data_yeast1	0.3368	0.5281

Tabla 6. Resultado Naive Bayes

Resultados

En definitiva, los resultados obtenidos con lo anteriormente expuesto más algunos otros con los que se hicieron pequeñas pruebas quedan resumidos en la siguiente tabla:

	Técnica							
Dataset	Standard						Min/Max	PCA
	SVM	Random Forest	XG Boost	Naive Bayes	Multiperceptron	Logistic Regression	XG Boost	Bagging
data_elephant	0,7514	0,8487	0,8233	0,718	0,7547	0,7558	0,8233	0,7309
data_rings	0,9811	0,9542	0,964	0,9811	-	0,7468	0,9633	0,7538
data_yeast1	0,7055	0,6979	0,7248	0,5281	-	0,6539	0,7148	-

Tabla 7. Resultados generales

Analizando la tabla, se consiguieron sacar las conclusiones que se muestran en el capítulo de Procesamiento general.

Conclusiones

Como conclusiones generales para el reto, cabe destacar que los tiempos de ejecución que se han obtenido, son menores que los 20 minutos que se dieron como máximo. Además, se puede decir, que los resultados obtenidos tanto con un dataset como con otro, son lo suficientemente óptimos ya que superan valores de 0.8 en la mayoría de los datasets. La siguiente tabla muestra los resultados tanto de AUC como de tiempos aplicando la función AutoML sin aplicar Voting.

Dataset	Best model	AUC	Hiperparámetros	Tiempos
data_elephants	Random Forest	0.868695270	'RF__n_estimators': 1200	288.836 s
data_rings	Ligth GBM	0.96299693	boosting_type: 1 colsample_bytree:0.6017 learning_rate: 0.6034 max_depth: 13.0 n_estimators: 35.0 num_leaves: 35.0 reg_lambda: 0.17898	731,193 s
data_yeast1	XGBoost	0.71271307	XGB__n_estimators: 100	39.670 s

Tabla 8. Tabla Resultados AutoML

Tras ejecutar la clase, pero habiéndole añadido el algoritmo de Voting Classifier, se espera resultados similares o mejores que la tabla anterior, ya que, como se sabe, el resultado final será el mejor entre la tabla anterior y este sistema de votos producido por los 3 mejores modelos. Los resultados son los siguientes, para cada uno de los 3 datasets que se nos proporcioban:

Dataset	Best model	AUC	Hiperparámetros	Tiempos
data_elephants	Random Forest	0.868695270	'RF__n_estimators': 1200	294.092 s
data_rings	Ligth GBM	0.96299693	boosting_type: 1 colsample_bytree:0.6017 learning_rate: 0.6034 max_depth: 13.0 n_estimators: 35.0 num_leaves: 35.0 reg_lambda: 0.17898	775.902 s
data_yeast1	XGBoost	0.71271307	XGB__n_estimators: 100	59.459 s

Tabla 9. Resultados finales con Voting Classifier

Se puede comprobar que, tanto en la primera tabla como en la segunda, lo único que cambia son los tiempos de ejecución. Por tanto, como conclusión final, se puede deducir que el modelo ha sido elegido correctamente sin tener la necesidad de aplicar Voting Classifier, ya que el valor de AUC y los hiperparámetros son los mismos. De esta manera, nos aseguramos que la decisión tomada ha sido correcta.

Además, no sabiendo cuáles serán los datos de entrada, cabe deducir que los árboles de decisión tomarán ventaja sobre el resto de algoritmos, ya que para los 3 que se tenían de ejemplo, los 3 modelos escogidos son algoritmos de árboles de decisión.

ANEXOS

El código se encuentra en el enlace de git hub: https://github.com/polazaro/DatathonMS

Anexo 1: def __init__()

```
def __init__(self):

        '''This is the constructor of the DatathonML class'''

        # All hyperparameters for the grid search
        SVM_parameters = {'SVM__C': [0.001, 0.01, 0.1, 10, 100, 10e5],
                          'SVM__gamma': [0.1, 0.01, 0.001]}
        RF_parameters = {'RF__n_estimators': [100, 300, 500, 800, 1000,
                                              1200]}

        XGB_parameters = {'XGB__n_estimators': [10, 30, 60, 80, 100, 120,
140,
                                                160, 180, 200, 220, 300,
                                                500, 800]}
        LR_parameters = {"LR__C": np.logspace(-3, 3, 7),
                         "LR__penalty": ["l1", "l2"]}
        KNN_parameters = {"KNN__n_neighbors": [3, 5, 7, 9, 11, 13, 15]}
        self.LGB_parameters = {
            'boosting_type': hp.choice('boosting_type', ['gbdt', 'dart']),
            'colsample_bytree': hp.uniform('colsample_by_tree', 0.6, 1.0),
            'learning_rate': hp.loguniform('learning_rate', np.log(0.01),
                                           np.log(1)),
            'max_depth': scope.int(hp.quniform('max_depth', 5, 15, 1)),
            'n_estimators': scope.int(hp.quniform('n_estimators', 5, 35, 1)),
            'num_leaves': scope.int(hp.quniform('num_leaves', 5, 50, 1)),
            'reg_lambda': hp.uniform('reg_lambda', 0.0, 1.0)}
        self.parameters = {'SVM': SVM_parameters,
                           'NB': {},
                           'RF': RF_parameters,
                           'XGB': XGB_parameters,
                           'LR': LR_parameters,
                           'LGB': self.LGB_parameters,
                           'KNN': KNN_parameters}

        # All models for training
        self.all_models = {'SVM': SVC(),
                           'NB': GaussianNB(),
                           'RF': RandomForestClassifier(random_state=15325),
                           'XGB': XGBClassifier(),
                           'LR': LogisticRegression(),
                           'LGB': lgb.LGBMClassifier(),
                           'KNN': KNeighborsClassifier()}

        # These are the dictionary needed to store the results
        self._results = {}
        self._best_parameters = {}

        print('Clase inicializada...')
```

Anexo 2: def read_dataframe()

```
def read_dataframe(self, dir_datasets, name):

        '''Function: This function read a .csv data into pandas dataframe

        Input:
                dir_datasets: directory where the data is stored as .csv
                name: name of the .csv that is going to be read
        Output:
                dataframe: a pandas dataframe'''

        try:
            dataframe = pd.read_csv(dir_datasets + name, sep=';',
                                    skipinitialspace=True)
        except:
            print('No se ha podido leer el dataframe correctamente')
            return []

        return dataframe
```

Anexo 3: def obtain_labels()

```
def __obtain_labels(self, df):

        '''Function: This function uses a label encoder to transform a
        categorical label into numerical one.

        Input:
                df: a pandas dataframe
        Output:
                label: all labels as list'''

        le = LabelEncoder()  # Label Encoder initialization
        label = le.fit_transform(list(df.iloc[:, -1]))

        return label
```

Anexo 4: def preprocess_dataframe()

```
def __preprocess_dataframe(self, df):

        '''Function: This function takes a dataframe as input and is able to
        preprocess and clean all data, first replacing the wrong characters and
        converting the type of the column, avoiding special errors. As well,
        this function erase the unique columns and separate data and labels.

            Input:
                    df: a pandas dataframe
            Output:
                    df_preprocessed: a pandasdataframe preprocessed as it
                    was explained before
                    labels: all labels as list'''

        # Erase the duplicate samples
        df = df.drop_duplicates()
```

```
        # Obtain labels
        labels = self.__obtain_labels(df)

        # Clean columns that are uniques and does not give us any information
        eliminate_columns = []
        df = df.iloc[:, :-1]  # Select dataframe except the label (last column)
        for column in df.columns:
            if df[column].dtype == 'object':  # Type replace
                df[column] = pd.to_numeric(df[column].str.replace(',', '.'),
                                           errors='coerce')
            if len(df[column].unique()) == 1:  # If the feature is unique
                eliminate_columns.append(column)
        df_preprocessed = df.drop(eliminate_columns, axis=1)

        return df_preprocessed, labels
```

Anexo 5: Correlación

```
def __calculate_corr(self, df):

        '''Function: This function uses the corr() function to obtain a matrix
        correlation of a pandas dataframe.

            Input:
                    df: a pandas dataframe
            Output:
                    correlation: matrix correlation'''

        correlation = df.corr()

        return correlation

def __eliminate_corr_features(self, df, threshold):

        '''Function: This function takes a dataframe as input and computes the
        correlation of the data. Than, using a threshold it drops the columns
        that are above this threshold, elminating randomly one of the two
        columns.

            Input:
                    df: a pandas dataframe
                    threshold: a threshold of correlation between features. It
                    is 0.8 by default
            Output:
                    df_no_correlation: dataframe without correlated feature
                    between them'''

        correlation = self.__calculate_corr(df)  # Computing matrix correlation
        columns = np.full((correlation.shape[0],), True, dtype=bool)
        for i in range(correlation.shape[0]):
            for j in range(i+1, correlation.shape[0]):
                if correlation.iloc[i, j] >= threshold:
                    if columns[j]:
                        columns[j] = False
        selected_columns = df.columns[columns]
        df_no_correlation = df[selected_columns]

        return df_no_correlation
```

Anexo 7: def compute_AUC ()

```
def __compute_auc(self, model, X_test, y_test):

        '''Function: This function computes the AUC for a set of data and a
        model given.

            Input:
                    model: scikit-learn model
                    X_test, y_test: data for testing the AUC
            Output:
                    AUC_metric: the AUC obtained with model'''

        fpr, tpr, thresholds = metrics.roc_curve(y_test, model.predict(X_test))
        AUC_metric = metrics.auc(fpr, tpr)

        return AUC_metric
```

Anexo 8: def training_function_bayes ()

```
def __training_function_bayes(self, param_space,
                                X_train, y_train, X_test, y_test, num_eval):

        '''Function: This function computes for the LGB classifier a bayesian
        optimization for the hyperparameters search. The output is kept in
        the class attributes to be used in other classes.

            Input:
                    X_train, y_train, X_test, y_test: training and testing data
                    num_eval: number of evaluations for the bayesian work
            Output:
                    None'''

        def objective_function(parameters):

            '''Internal function for the training_function_bayes'''

            clf = lgb.LGBMClassifier(**parameters)
            score = cross_val_score(clf, X_train, y_train, cv=5).mean()
            return {'loss': -score, 'status': STATUS_OK}

        # Bayesian optimization, the objective_function is the loss function
        trials = Trials()
        best_param = fmin(objective_function,
                          param_space,
                          algo=tpe.suggest,
                          max_evals=num_eval,
                          trials=trials,
                          rstate=np.random.RandomState(1))

        # Obtaining the best parameters
        best_param_values = [x for x in best_param.values()]
        if best_param_values[0] == 0:
            boosting_type = 'gbdt'
        else:
            boosting_type = 'dart'

        # Selecting the best model of the bayesian optimization
```

```
        best_model = lgb.LGBMClassifier(learning_rate=best_param_values[2],
                                        num_leaves=int(best_param_values[5]),
                                        max_depth=int(best_param_values[3]),
                                        n_estimators=int(best_param_values[4]),
                                        boosting_type=boosting_type,
                                        colsample_bytree=best_param_values[1],
                                        reg_lambda=best_param_values[6])

        # Fitting the best model
        best_model.fit(X_train, y_train)

        # Saving the best result
        self._results['LGB'] = self.__compute_auc(best_model, X_test, y_test)
        parameters = list(self.LGB_parameters.keys())
        self._best_parameters['LGB'] = [(parameters[i], best_param_values[i])
                                        for i in range(len(parameters))]
```

Anexo 9: def select_best_models ()

```
def __select_best_models(self, all_models, range_p, flag):

        '''Function: This function select the best models in two different
        ways. This first one is when the flag = 0, where after sorting all the
        models AUC, it chooses the best model and the other ones that are at
        most a range_p far away from the best one. When flag=1, it just takes
        the best model and restart the main attributes of the class.

            Input:
                    all_models: dictionary with the name of the model and the
                    AUC values
                    range_p: percentage/100 of the maximum distance between the
                    best AUC and the other ones that are going to be considered
                    flag: flag that takes two different ways
            Output:
                    final_models: a list with the best models or the best
                    model'''

        # Sorting the dictionary by value (AUC)
        best_models_sort = {k: v for k, v in sorted(all_models.items(),
                                                    key=lambda item: item[1],
                                                    reverse=True)}

        if flag == 0:  # For the first approximation

            final_models = ['SVM']
            max_value = list(best_models_sort.values())[0]
            for model in best_models_sort:
                if best_models_sort[model] >= (max_value-range_p):
                    final_models.append(model)

        elif flag == 1:  # For the final selection

            final_models = [list(best_models_sort)[0],
                            list(best_models_sort.values())[0],
                            self._best_parameters[list(best_models_sort)[0]]]
            self._best_parameters = {}
            self._results = {}
```

```
        else:  # Take the first three elements (best 3 models)
            try:
                auxiliar = [(model, self._best_parameters[model])
                            for model, aux in
                            list(best_models_sort.items())[:3]]
                final_models = []
                for model, params in auxiliar:
                    parameter_changed = {}
                    for param in params:
                        parameter_changed[param.split('__')[1]] = params[param]
                    final_models.append((model, parameter_changed))
            except:
                final_models = []

        return final_models
```

Anexo 10: def first_approximation_training ()

```
def __first_approximation_training(self, X_train, X_test, y_train, y_test):

        '''Function: This function computes with the default hyperparameters
        the AUC of each model..

            Input:
                    X_train, y_train, X_test, y_test: training and testing data
            Output:
                    best models: dictionary with the name of the model and the
                    AUC value'''

        best_models = {}
        for name in self.all_models:
            model = self.all_models[name]
            model.fit(X_train, y_train)
            best_models[name] = self.__compute_auc(model, X_test, y_test)

        return best_models
```

Anexo 11: def training_function_grid ()

```
def __training_function_grid(self, X_train, X_test, y_train, y_test,
                             parameters, model, type_clf):

        '''Function: This function computes a GridSearch for the model passed
        by parameter. It saves the results in the main attributes of the class.

            Input:
                    X_train, y_train, X_test, y_test: training and testing data
                    parameters: possible hyperparameters for the grid search
                    model: scikit-learn model
                    type_clf: name of the model
            Output:
                    None'''

        steps = [('scaler', StandardScaler()),
                 ('percentile', SelectPercentile(f_classif, percentile=75)),
                 (type_clf, model)]
        pipeline = Pipeline(steps)
```

```
        grid = GridSearchCV(pipeline, param_grid=parameters, cv=5)

        grid.fit(X_train, y_train)
        self._results[type_clf] = self.__compute_auc(grid, X_test, y_test)
        self._best_parameters[type_clf] = grid.best_params_
```

Anexo 12: def __ensemble_voting ()

```
def __ensemble_voting(self, X_train, X_test, y_train, y_test, models):

        '''Function: This function takes the best three models and create an
        ensemble model with majority voting.

            Input:
                    X_train, y_train, X_test, y_test: training and testing data
                    parameters: possible hyperparameters for the grid search
                    model: best scikit-learn model
            Output:
                    None'''

        # Creating the models with the best parameters
        clf1 = self.all_models[models[0][0]].set_params(**models[0][1])
        clf2 = self.all_models[models[1][0]].set_params(**models[1][1])
        clf3 = self.all_models[models[2][0]].set_params(**models[2][1])
        v_ensembl = VotingClassifier(estimators=[(models[0][0], clf1),
                                                 (models[1][0], clf2),
                                                 (models[2][0], clf3)],
                                     voting='hard')

        # Fitting the model and creating the final result
        v_ensembl.fit(X_train, y_train)
        type_clf = ('Ensemble voting with: ' +
                    models[0][0] + '__' + models[1][0] + '__' + models[2][0])
        self._results[type_clf] = self.__compute_auc(v_ensembl, X_test, y_test)
```

Anexo 13: def __print_final_solution ()

```
def __print_final_solution(self, final_solution):

        '''Function: This function prints the results of the best model and the
        hyperparameters selected by the grid search.

            Input:
                    final_solution: a list with the name, the AUC and the
                    hyperparamets of the best model
            Output:
                    None'''

        print('\n######### SOLUCIÓN #########\n')
        print('Best model: ' + final_solution[0])
        print('AUC = ' + str(final_solution[1]))
        print('Grid Search parameters: ')
        print(final_solution[2])
        print('\n############################\n')
```

Anexo 14: def auto_ml ()

```
def auto_ml(self, df):


        ### IDEA: HACER TODOS LOS MODELOS CON UN PARÁMETRO POR DEFECTO Y ELIMINAR
        ### AQUELLOS QUE DEN BAJOS. QUEDARSE CON LOS 3 MÁS ALTOS Y HACER GRID_SEARCH

        ### Preprocessing dataframes
        data, labels = self.preprocess_dataframe(df)

        ### Eliminating correlated features from dataset
        data_nocorr = self.eliminate_corr_features(data,
                                                   self.calculate_corr(data),
                                                   0.8)

        ### Data for training
        X = data_nocorr
        Y = labels

        ### Splitting for training and test
        X_train, X_test, y_train, y_test = train_test_split(X,Y,test_size=0.3,
                                                            random_state=30,
                                                            stratify=Y)


        ### First approximation to eliminate the worst models
        first_approach = self.first_approximation_training(X_train, X_test,
                                                           y_train, y_test)

        print('First approach: ', first_approach)

        best_models = self.select_best_models(first_approach, 0.15, 0)

        print('Models from first approach: ', self.all_possible_models)
        for name_model in self.all_possible_models:

            print(name_model)
            ### Training all possible models
            if name_model != 'LGB':
                self.training_function_grid(X_train, X_test,
                                            y_train, y_test,
                                self.parameters[name_model],
                                self.all_possible_models[name_model],
                                name_model)
            else:
                self.training_function_bayes(self.parameters[name_model], X_train,
                                             y_train, X_test,
                                             y_test, 75)


        return self.select_best_models(self.results, 0, 1)
```