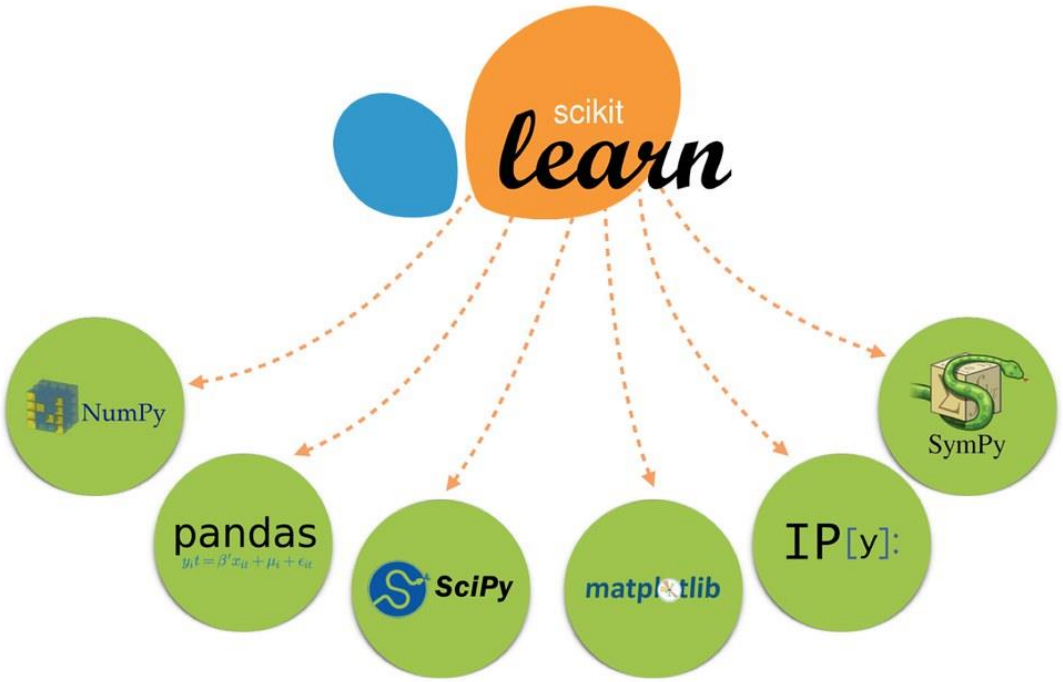
Introducción a Scikit Learn



https://anderfernandez.com/blog/tutorial-sklearn-machine-learning-python/

Scikit Learn es una librería de Machine Learning en Python que busca ayudarnos en los principales aspectos a la hora de afrontar un problema de Machine Learning. Más concretamente, Scikit Learn cuenta con funciones para ayudarnos en:

- Preprocesamiento de los datos, incluyedno:
    - Split entre train y test.
    - Imputación de valores perdidos.
    - Transformación de los datos.
    - Feature engineering.
    - Feature selection.
- Creación de modelos, incluyendo:
    - Modelos supervisados
    - Modelos no supervisados
- Optimización de hiperparámetros de los modelos

Como ves, Scikit Learn es una librería muy completa y muy útil (por ello es tan conocida y utilizada). Para ver todas estas funcionalidades vamos a necesitar unos datos. Para ello usaré el módulo datasets de sklearn para poder usar algunos de los datasets que trae la librería por defecto:

Empecemos con nuestro tutorial de Scikit Learn viendo la lógica detrás de Scikit learn.

```
from sklearn import datasets
import pandas as pd
import numpy as np

wine = datasets.load_wine()

data = pd.DataFrame(data= wine['data'],
                    columns= wine['feature_names'])

y = wine['target']
print(y[:10])
data.head()
```

Lógica detrás de Sklearn

Una cuestión muy interesante y útil de Sklearn es que, tanto en la **preparación de los datos** como en la creación del modelo hace una **distinción entre train y transform o predict.**

Esto es algo muy interesante, ya que nos **permite guardar estos ficheros de train** para que, a la hora de hacer las transformaciones o la predicción simplemente haya que cargar ese fichero y hacer la transformación/predicción.

Así pues, cuando trabajemos con Sklearn, tendremos que acostumbrarnos a **primero hacer el train y después ejecutarlo** sobre nuestros datos (se puede hacer todo de un paso, pero yo siempre prefiero separarlos).

Preprocesamiento de datos con Sklearn

Split entre train y test

Como ya sabrás, antes de abarcar cualquier transformación en nuestro dataset, debemos primero dividir nuestros datos entre train y test. La idea es que los datos de test no se tengan considerados en niguna transformación, como si fuesen nuevos de verdad.

Así pues, para realizar el split entre **train y test** contamos con la función train_test_split, la cual **devuelve una tupla con cuatro elementos: Xtrain, Xtest, Ytrain e Ytest.**

Asimismo, para que el split sea reproducible **podemos fijar la semilla** usando el parámetro random_state.

Veamos cómo funciona:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(data, y,
                                    test_size = 0.8,
                                    random_state = 1234)

print(f'X train shape {X_train.shape} \nX test shape {X_test.shape}')
X train shape (35, 13)
X test shape (143, 13)
```

Como ves hacer el split con sklearn es super sencillo.

Imputación de Valores Perdidos con Sklearn

Lo primero de todo, vamos a comprobar si nuestro **dataset contiene valores perdidos** de tal forma que podamos imputarlos:

```
X_train.isna().sum()
```

```
alcohol                         0
malic_acid                      0
ash                             0
alcalinity_of_ash               0
magnesium                       0
total_phenols                   0
flavanoids                      0
nonflavanoid_phenols            0
proanthocyanins                 0
color_intensity                 0
hue                             0
od280/od315_of_diluted_wines    0
proline                         0
dtype: int64
```

Como vemos, el dataset no contiene ningún valor perdido, pero no pasa nada. Vamos a usar una copia de este dataset y **crear unos Na para demostrar cómo funciona** la imputación de valores perdidos en Sklearn:

```
data_na = X_train.copy()

for col in data_na.columns:
   data_na.loc[data_na.sample(frac=0.1).index, col] = np.nan
data_na.isna().sum()
```

Cuando tenemos valores perdidos, hay varios enfoques que podemos seguir:

- **Eliminar las observaciones** (poco recomendado, sobre todo si tenemos pocos datos).
- **Imputar un valor constante** obtenida de la propia variable (**la media, moda, mediana,** etc.) A este tipo de imputación se le conoce como **imputación univariante**.
- **Usar todas las variables disponibles para hacer la imputación**, es decir, imputación multivariante. **Un modelo típico de imputación multivariante es el uso de un modelo kNN.**
-

Todas estas opciones las tenemos dentro del modulo impute de Sklearn. Sigamos con el tutorial de Sklearn, viendo cómo funciona la imputación simple.

Imputación de valores perdidos univariantes

Dentro de la imputación univariante tenemos varios valores que podemos imputar, más concreatamente puedes imputar la media, la mediana, la moda o un valor fijo. Personalmente no me gusta imputar la media, puesto que puede verse muy afectada por la distribución de los datos (los outliers puede afectar mucho a la media). En su lugar suelo preferir otros valores como la moda o la mediana ya que son menos sensibles a outliers.

Veámos cómo podemos hacer la **imputación univariante** en Sklearn:

```
# Imputación Univariante
from sklearn.impute import SimpleImputer

mode_imputer = SimpleImputer(strategy = 'most_frequent')

# For each column, make imputation
for column in data_na.columns:
   values = data_na[column].values.reshape(-1,1)
   mode_imputer.fit(values)
   data_na[column] = mode_imputer.transform(values)

# Check Nas
data_na.isna().sum()
```

Como has podido ver, de una forma muy sencilla hemos podido **imputar absolutamente todos los valores perdidos que teníamos en el dataset con la moda** de una forma muy sencilla.

Además, para hacer la imputación con otro valor como la media o la mediana simplemente habría que cambiar la estrategia por mean o por median, respectivamente.

Como ves, imputar valores perdidos usando los datos de la propia variable es muy sencillo con Sklearn. Sin embargo, Sklearn va mucho más allá y ofrece otras cuestiones como la imputación teniendo en cuenta varias variables. Veamos cómo funciona.

Imputación multivariante de valores perdidos

La idea detrás de una **imputación multivariante es crear un regresor** e intentar predecir cada una de las variables con el resto de variables que tenemos. De esta forma, **el regresor puede aprender la relación entre los datos y podrá realizar una imputación usando todas las variables del dataset**.
Esta se trata de una característica que aún está en fase experimental en Sklearn. Es por ello que, para que funcione, primero tendremos que habilitarla importando enable_iterative_imputer.

```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

data_na = X_train.copy()

# Generate new nas
for col in data_na.columns:
    data_na.loc[data_na.sample(frac=0.1).index, col] = np.nan

# Create imputer
iter_imputer = IterativeImputer(max_iter=15, random_state=1234)

# Transform data
iter_imputer_fit = iter_imputer.fit(data_na.values)
imputed_data = iter_imputer_fit.transform(data_na)

pd.DataFrame(imputed_data, columns = data_na.columns)\
    .isna()\
    .sum()
```

Como ves, hemos **creado un sistema de imputación que tiene en cuenta todas las variables de cara a realizar la imputación de los valores perdidos**.
Asimismo, dentro de los imputadores multivariantes, una forma muy típica de realizar la imputación es usando el modelo kNN. Esto es algo que Sklearn también ofrece. Veámoslo.

Imputación mediante kNN

De cara a la imputación de valores perdidos usando el algoritmo kNN, **Sklearn busca las observaciones que sean más parecidas para cada observación con valores perdidos, y usa los valores de esas observaciones para hacer la imputación**.

Al igual que en el algoritmo kNN normal, el único parámetro que podemos modificar es el número de vecinos a tener en cuenta para hacer la predicción.

Existen dos formas de elegir el número de vecinos:

- La raíz cuadrada de observaciones, asegura que no eliges un valor ni demasiado pequeño ni demasiado grande.
- Método del codo. Consiste en calcular el error para diferentes valores de *k* y elegir aquél valor que lo minimice.

En este caso, al ser una prueba usaré la aproximación de la raíz cuadrada.
Así pues, para imputar los valores perdidos con Sklearn usando kNN tendremos que usar la función KNNImputer.

```
from sklearn.impute import KNNImputer

data_na = X_train.copy()

# Generate new nas
for col in data_na.columns:
    data_na.loc[data_na.sample(frac=0.1).index, col] = np.nan

# Select k
k = int(np.round(np.sqrt(data_na.shape[0])))

# Create imputer
knn_imputer = KNNImputer(n_neighbors=k)

# Transform data
knn_imputer_fit = knn_imputer.fit(data_na.values)

imputed_data = knn_imputer_fit.transform(data_na)

pd.DataFrame(imputed_data, columns = data_na.columns)\
    .isna()\
    .sum()
```

Con esto ya hemos visto la imputación de valores perdidos. Ahora veáms cómo hacer la transformación de los datos con Sklearn.

Transformación de los datos

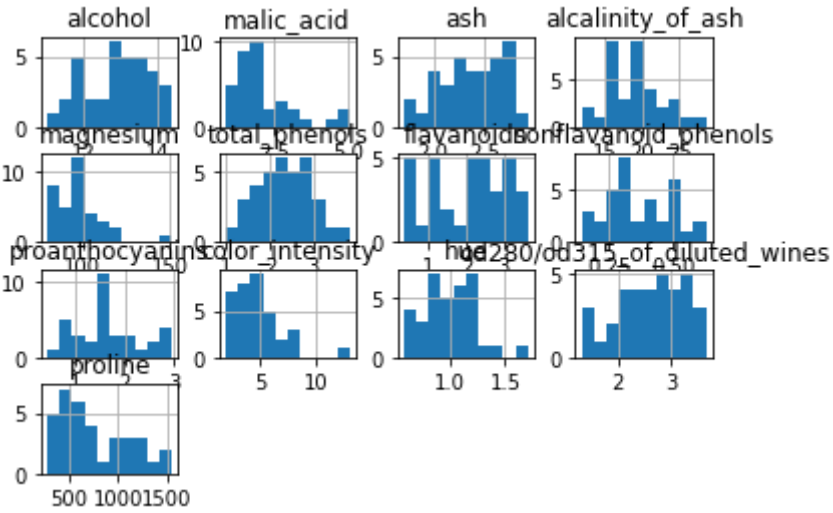
Son muchas las transformaciones que podemos y a veces debemos aplicar a nuestros datos, tales como: normalización, transformaciones para seguir una distribución, One-Hot encoding…

Para ello Sklearn ofrece el módulo de preprocesado, gracias al cual podemos realizar todas las transformaciones comentadas anteriormente y más. Así pues, veamos cómo son los datos que tenemos para, a partir de ahí, ponernos a transformar los datos:

```
X_train.describe()
```

```
%matplotlib inline

X_train.hist()
```



Como vemos, tenemos **varias cuestiones interesantes que, seguramente, tendremos que transformar, tales como:**

- Variables escoradas a la izquierda.
- Variables escoradas a la derecha.
- Variables con posibles outliers.

Veámos cada uno de estas cuestiones yendo poco a poco. Empecemos con cómo modificar las distribuciones de los datos.

Modificar la distribución de una variable

Cojamos el ejemplo de la variable malic_acid la cual está claramente escorada a la izquierda. En estos casos, Sklearn dentro del módulo de preprocessing ofrece las funciones QuantileTransformer y PowerTranformer con las que podemos evitar el skewness de nuestros datos. Veamos cómo funcionan.

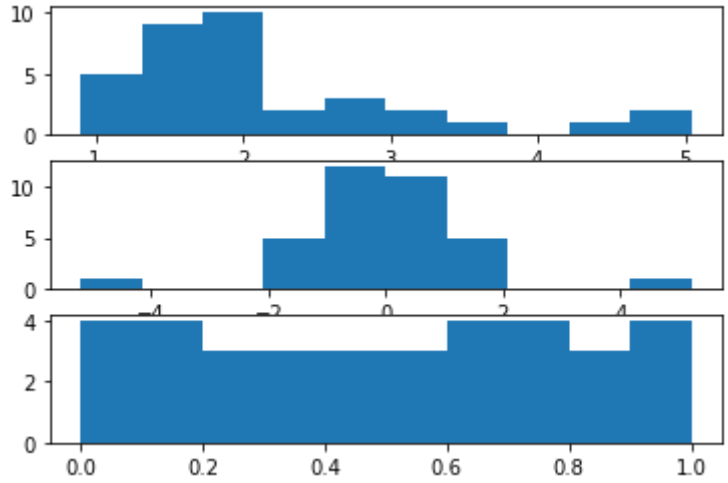
```
from sklearn import preprocessing
import matplotlib.pyplot as plt

quantile_transf_norm =  preprocessing.QuantileTransformer(output_distribution= 'normal')
quantile_transf_uniform =  preprocessing.QuantileTransformer(output_distribution=
'uniform')

data_to_transform = X_train['malic_acid'].values.reshape(-1,1)
transformed_data_normal =  quantile_transf_norm.fit_transform(data_to_transform)
transformed_data_uniform =  quantile_transf_uniform.fit_transform(data_to_transform)

# Create plots
fig, axs = plt.subplots(3)

axs[0].hist(X_train['malic_acid'].values)
axs[1].hist(transformed_data_normal)
axs[2].hist(transformed_data_uniform)
```



Como puedes ver, hemos pasado de una variable escorada a la izquierda a una variabe que sigue una distribución normal o que sigue una distribución uniforme gracias a la función QuantileTransformer de Sklearn.

Claramente la transformación a aplicar dependerá del caso en concreto, pero como ves, una vez sabemos que tenemos que transformar los datos, hacerlo con Sklearn es muy sencillo.

Así pues, sigamos con nuestro tutorial de Sklearn, viendo cómo normalizar o estandarizar las variables.

Cómo normalizar o estandarizar los datos en Sklearn

Otras transformaciones típicas que podemos aplicar son la normalización y estandarización, las cuales podemos realizar con las funciones StandardScaler y MinMaxScaler, respectivamente.
En mi opinión, suele ser preferible estandarizar que normalizar, puesto que la normalización puede generar problemas en producción. En cualquier caso, veámos cómo podemos normalizar y estandarizar en Python con Sklearn:

```
# Create the scaler
stand_scale = preprocessing.StandardScaler()
normal_scale = preprocessing.MinMaxScaler()

# Fit the scaler
stand_scale_fit = stand_scale.fit(X_train)
normal_scale_fit = normal_scale.fit(X_train)

# Apply the scaler to train
X_train_scale = stand_scale_fit.transform(X_train)
X_train_norm = normal_scale_fit.transform(X_train)

# Apply the scaler to test
X_test_scale = stand_scale_fit.transform(X_test)
X_test_norm = normal_scale_fit.transform(X_test)

# Convert data to DataFrame
X_train_scale = pd.DataFrame(X_train_scale, columns = data_na.columns)
X_train_norm = pd.DataFrame(X_train_norm, columns = data_na.columns)

# Check Standardization
standard_sd =  X_train_scale['malic_acid'].std()
standard_mean = X_train_scale['malic_acid'].mean()
normalized_min =  X_train_norm['malic_acid'].min()
normalized_max = X_train_norm['malic_acid'].max()

print(f'Standardized data has SD of {np.round(standard_sd)} and mean of
{np.round(standard_mean)}')
print(f'Normalized data has Min of {normalized_min} and max of {normalized_max}')
```

Como ves, aplicar normalizar o estandarizar los datos con Sklearn es muy sencillo. Sin embargo, estas transformaciones únicamente aplican a los datos numéricos. Ahora, veamos cómo hacer one-hot encoding de los datos, que es la transformación principal de los datos categóricos.

Cómo hacer One-hot encoding con Sklearn

Cuando trabajamos con variables categóricas, una de las cuestiones más importantes es transformar nuestras variables categóricas en numéricas. Para ello, aplicamos la dumificación o one-hot encoding, la cual consiste en crear tantas nuevas variables menos una como opciones tiene la variable y darle valor de 1 o 0.

En este sentido, resultado fundamental realizar el proceso de One-hot encoding después de haber hecho las transformaciones a variables numéricas (normalización, estandarización, etc.). Esto es debido a que, sino, estaremos transformando estas variables y dejarán de tener sentido.

Así pues, realizar una transformación de One-hot encoding con Sklearn de forma muy sencilla gracias a la función OneHotEncoder. Para ver cómo funciona, creare una lista con tres posibles valores: UK, USA o Australia.

```
label = np.array(['USA','Australia', 'UK', 'UK','USA','Australia','USA',
                  'UK', 'UK','UK','Australia'])

label
array(['USA', 'Australia', 'UK', 'UK', 'USA', 'Australia', 'USA', 'UK',
       'UK', 'UK', 'Australia'], dtype='<U9')
```

Así pues, para codificar la variable es tan sencillo como pasar la variable a la función OneHotEncoder. Sin embargo, por defecto esta función crea tantas variables como posibles opciones hay. Esto no suele ser una buena idea, ya que con n-1 opciones sería suficiente. Así pues, con el parámetro drop='first' podemos evitar redundancias en los datos.

Además, algo típico a la hora de poner un modelo en predicción es que aparezca un nuevo nivel que no estaba contemplado en el entrenamiento. Por defecto, esto generará un error, lo cual puede que no sea idóneo (sobre todo si se trata de un modelo en batch sobre varios datos). Para evitar que haya problemas si esto ocurre podemos usar fijar indicar handle_unknown = 'ignore'.

Veamos cómo hacerlo:

```
from sklearn.preprocessing import OneHotEncoder

# Create encoder
ohencoder = OneHotEncoder(drop='first')

# Fit the encoder
ohencoder_fit = ohencoder.fit(label.reshape(-1,1))

# Transform the data
ohencoder_fit.transform(label.reshape(-1,1)).toarray()
array([[0., 1.],
       [0., 0.],
       [1., 0.],
       [1., 0.],
       [0., 1.],
       [0., 0.],
       [0., 1.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [0., 0.]])
```

Como ves, transformar nuestros datos con Sklearn es algo súper sencillo. Además, eso no es todo, donde más provecho podemos sacar a Sklearn (y por lo que es más conocido) es en la creación de modelos de Macine Learning.
Sigamos con nuestro tutorial de Sklearn, viendo cómo crear modelos de Machine Learning.

Cómo crear modelos de Machine Learning con Sklearn

Dentro de Sklearn contamos con muchísimas familias de modelos de Machine Learning diferentes que podemos aplicar y, dentro de cada familia, puede haber varios modelos diferentes.

Poe lo tanto, en la siguiente tabla te incluyo todas las familias de modelos de Machine Learning junto con el nombre del módulo donde se encuentran:

Modelo Supervisado	**Module**
Linear Models	linear_model
Linear and Quadratic Discriminant Analysis	discriminant_analysis
Kernel ridge regression	kernel_ridge
Support Vector Machines	svm
Stochastic Gradient Descent	linear_model
Nearest Neighbors	neighbors
Gaussian Processes	gaussian_process
Cross decomposition	cross_decomposition
Naive Bayes	naive_bayes
Decision Trees	tree
Ensemble methods	ensemble
Multiclass and multioutput algorithms	multiclass & multioutput
Semi-supervised learning	semi_supervised
Isotonic regression	isotonic
Probability calibration	calibration
Neural network models (supervised)	neural_networ

En el caso de los modelos no supervisados pasa un poco parecido ya que cuenta con muchos modelos no supervisados que podemos encontrarn en diferentes módulos:

Model	**Module**
Gaussian mixture models	GaussianMixture
Manifold learning	manifold
Clustering	cluster
Decomposing signals in components (matrix factorization problems)	decomposition
Covariance estimation	covariance
Neural network models (unsupervised)	neural_network

Si bien en este post no vamos a poder cubrir todos los modelos de Machine Learning, sí que vamos a usar algunos modelos principales. Más concretamente, vas a aprender cómo:

1. Crear un modelo de Machine Learning con Sklearn
2. Valida el rendimiento de tu modelo con Sklearn.
3. Encontrar los valores óptimos de los hiperparámetros del modelo.

Considerando lo anterior, veamos cómo funciona:

Cómo crear un modelo de Machine Learning con Sklearn

Para crear un modelo de machine learning con Sklearn, primero tenemos que saber qué modelo queremos crear, puesto que, como hemos visto anteriormente, cada modelo puede que se encuentre en un módulo difernte.
En nuestro caso vamos a crear dos modelos de clasificación: regresión logística y Random Forest. Como en los casos anteriores, lo primero de todo vamos a crear nuestros modelos, primero haciendo el fit en train y después el predict sobre los daatos de test.

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier



## -- Logistic Regression -- ##

# Create the model
log_reg = LogisticRegression(penalty = 'none')

# Train the model
log_reg_fit = log_reg.fit(X_train, y_train)

# Make prediction
y_pred_log_reg = log_reg_fit.predict(X_test)



## -- Random Forest -- ##

# Create the model
rf_class = RandomForestClassifier()

# Train the model
rf_class_fit = rf_class.fit(X_train, y_train)

# Make prediction
y_pred_rf_class = rf_class_fit.predict(X_test)

print(f'Logistic Regression predictions {y_pred_log_reg[:5]}')
print(f'Random Forest predictions {y_pred_rf_class[:5]}')
```

Standardized data has SD of 1.0 and mean of -0.0
Normalized data has Min of 0.0 and max of 1.0

Como vemos, hemos creado los modelos de una forma muy sencilla. Ahora tendremos que evaluar cómo de buenos son nuestros modelos. Veamos cómo funciona.

Cómo medir el rendimiento de un modelo en Sklearn

La forma de evaluar el rendimiento de un modelo es analizar cómo de buenas son sus predicciones. Para ello Sklearn cuenta con diferentes funciones dentro del módulo metrics de Sklearn.
En nuestro caso, al tratarse de un modelo de clasificación podemos usar métricas como la matríz de confusión o el accuracy, entre otros. Veamoslo.

```
from sklearn.metrics import confusion_matrix, precision_score, recall_score, accuracy_score

print('##-- Logistic Regression --## ')
print(f'Acuraccy: {accuracy_score(y_test, y_pred_log_reg)}')
print(f'Precision: {precision_score(y_test, y_pred_log_reg, average="macro")}')
print(f'Recall: {recall_score(y_test, y_pred_log_reg, average="macro")}')
print(confusion_matrix(y_test, y_pred_log_reg))



print('\n##-- Random Forest --## ')
print(f'Acuraccy: {accuracy_score(y_test, y_pred_rf_class)}')
print(f'Precision: {precision_score(y_test, y_pred_rf_class, average="macro")}')
print(f'Recall: {recall_score(y_test, y_pred_rf_class, average="macro")}')
print(confusion_matrix(y_test, y_pred_rf_class))
```

```
##-- Logistic Regression --##
Acuraccy: 0.7202797202797203
Precision: 0.7578378120830952
Recall: 0.7104691075514874
[[40  6  0]
 [12 42  3]
 [14  5 21]]

##-- Random Forest --##
Acuraccy: 0.965034965034965
Precision: 0.9636781090033123
Recall: 0.9682748538011695
[[46  0  0]
 [ 2 53  2]
 [ 0  1 39]]
```

Como vemos, el modelo Random Forest ha tenido un mucho mejor resultado que el modelo de regresión logística. Sin embargo, nosotros no hemos tocado ninguno de los hiperparámetros del Random Forest. Es posible que, buscando los parámetros óptimos tengamos un mejor resultado aún.
Así pues, sigamos con el tutorial de Sklearn viendo cómo encontrar los hiperparámetros de un modelo.

Cómo tunear un modelo con Sklearn

De cara a encontrar los valores óptimos de los hiperparámetros de un modelo, Sklearn ofrece una función muy útiles: GridSearchCV.
Esta función permite definir un diccionario de parámetros, definiendo para cada parámetro todos los valores que se quieren probar. Además, para cada estimación de cada parámetro podemos hacer Cross Validation, de tal forma que no haya problemas de overfitting.
Así pues, simplemente con pasar nuestro modelo y los parámetros que queremos que compruebe a la función GridSearchCV y hacer el grid, Sklearn hará el grid search para todas las combinaciones posibles de todos los hiperparámetros y, por defecto, lo comprobará con 5 cross validations.
De este resultado podremos obtener el valor obtenido por todas las combinaciones con la clase cv_results_.
Así pues, probemos un montón de valores posibles para nuestro Random Forest y veámos cómo funciona el modelo:

```
from sklearn.model_selection import GridSearchCV, RepeatedKFold

rf_class = RandomForestClassifier()
grid = {
   'max_depth':[6,8,10],
   'min_samples_split':[2,3,4,5],
   'min_samples_leaf':[2,3,4,5],
   'max_features': [2,4,6,8,10]
   }

rf_class_grid = GridSearchCV(rf_class, grid, cv = 10)
rf_class_grid_fit =  rf_class_grid.fit(X_train, y_train)

pd.concat([pd.DataFrame(rf_class_grid_fit.cv_results_["params"]),
        pd.DataFrame(rf_class_grid_fit.cv_results_["mean_test_score"],
                columns=["Accuracy"])],axis=1)
```

Como puedes ver obtenemos el accuracy del modelo para diferentes vaores de los hiperparámetros y, como cabe de esperar, diferentes combinaciones de hiperparámetros generan diferentes resultados. Los modelos con un max_depth muy alto, por ejemplo, parecen funcionar peor que aquellos con un max_depth más bajo.

Ahora que hemos hecho Grid Search podemos encontrar cuál es el mejor modelo de todo los que hemos probado:

```
print(f'Best parameters: {rf_class_grid_fit.best_params_}')
print(f'Best score: {rf_class_grid_fit.best_score_}')
Best parameters: {'max_depth': 6, 'max_features': 2, 'min_samples_leaf': 2,
'min_samples_split': 2}
Best score: 1.0
```

Ahora que tenemos nuestro modelo entrenado, podemos hacer la predicción sobre test, a ver qué resultado conseguimos.

```
y_pred_rf_grid = rf_class_grid_fit.predict(X_test)

print('##-- Random Forest Grid Search & CV --## ')
print(f'Acuraccy: {accuracy_score(y_test, y_pred_rf_grid)}')
print(f'Precision: {precision_score(y_test, y_pred_rf_grid, average="macro")}')
print(f'Recall: {recall_score(y_test, y_pred_rf_grid, average="macro")}')
print(confusion_matrix(y_test, y_pred_rf_grid))
```

```
##-- Random Forest Grid Search & CV --##
Acuraccy: 0.972027972027972
Precision: 0.9714617554338809
Recall: 0.9766081871345028
[[46  0  0]
 [ 3 53  1]
 [ 0  0 40]]
```

Como veis, con el tuning hemos de los hiperparámetros hemos conseguido que nuestro modelo pase de una tasa de acierto del 95,8% a una tasa del 97,2% y todo ello de una forma súper sencilla.
Pero, aunque parezca mentira, aún hay más. Y es que, para mí, una de las cosas más chulas de Sklearn es que permite juntar todo el proceso de Machine Learning en un mismo paso. Sigamos con nuestro tutorial de Sklearn y veamos cómo funcionan los pipelines

Creando un pipeline de Machine Learning

Si te fijas, el proceso que hemos seguido hasta ahora ha sido un proceso bastante secuencial: primero eliminas variables, las estandarizas, dumificas, entrenas el modelo y haces la predicción.

Hasta ahora, cada uno de estos pasos ha ido por separado, de tal forma que primero siempre declaras la transformación el modelo, después haces el fit (o train en caso del modelo) y por último terminas aplicándolo.

Por suerte, Sklearn ofrece una forma mucho más sencilla de hacer todo este proceso: los pipelines y los Columns Transformers. Gracias a los pipelines y Column Transformers, en vez de tener que hacer cada paso por separado, podremos definir qué pasos queremos que se hagan y el propio Sklearn aplicará todos los pasos que le indiquemos de manera secuencial.

La única diferencia entre los pipelines y los column transformers es que un pipeline permite aplicar varias operaciones sobre una columna pero sin que se pueda paralelizar. Sin embargo, el ColumnTrasnformer solo sirve para aplicar una única operación por columna, pero se puede paralelizar.
En nuestro caso, como hemos realizado varias operaciones sobre una misma columna, usaremos el pipeline. Veamos cómo usarlo:

```
from sklearn.pipeline import Pipeline
from sklearn.feature_selection import VarianceThreshold

pipe = Pipeline([
   ('scaler', preprocessing.StandardScaler()),
   ('selector', VarianceThreshold()),
   ('classifier', RandomForestClassifier(max_depth=6,
                                max_features= 2,
                                min_samples_leaf = 2,
                                min_samples_split = 3
                                ))
   ])



pipe_fit = pipe.fit(X_train, y_train)
y_pred_pipe = pipe_fit.predict(X_test)

print('##-- Random Forest with Pipe --## ')
print(f'Acuraccy: {accuracy_score(y_test, y_pred_pipe)}')
print(f'Precision: {precision_score(y_test, y_pred_pipe, average="macro")}')
print(f'Recall: {recall_score(y_test, y_pred_pipe, average="macro")}')
print(confusion_matrix(y_test, y_pred_pipe))
```

```
##-- Random Forest with Pipe --##
Acuraccy: 0.972027972027972
Precision: 0.9702380952380952
Recall: 0.9766081871345028
[[46  0  0]
 [ 2 53  2]
 [ 0  0 40]]
```

Como ves, hemos creado un modelo haciendo toda la transformación del modelo en unas pocas líneas de código. Además, Sklearn ofrece la opción de guardar el modelo y el pipeline, de tal forma que la puesta en producción del modelo sea algo bastante sencillo. Veamos cómo guardar un pipeline:

```
from joblib import dump, load
from datetime import datetime

dump(pipe_fit, 'models/pipeline.joblib')

# Remove element and reload it
del(pipe_fit)

try:
   pipe_fit
except NameError:
   print(f'{datetime.now()}: Pipe does not exist.')

# Reload pipe
pipe_fit = load('models/pipeline.joblib')

try:
   pipe_fit
   print(f'{datetime.now()}: Pipe is loaded.')
except NameError:
   print(f'{datetime.now()}: Pipe not defined.')
```

```
2021-10-10 13:27:21.377056: Pipe does not exist.
2021-10-10 13:27:26.120434: Pipe is loaded.
```

Ahora, podremos usar ese pipeline para poder hacer la transformación sobre nuevos datos:

```
pipe_fit.predict(X_test)[:10]

array([1, 1, 1, 1, 2, 1, 2, 0, 0, 2])
```

Asimismo, supongamos que existiese una columna categórica, sobre la cual nos gustaría aplicar One Hot Encoding, como hemos visto anteriormente. En este caso, podríamos crear:

1. Un pipeline para las variables numéricas.
2. Otro pipeline para las variables categóricas.
3. Un ColumnTransformer que aplique el pipeline de las variables numéricas a las variables numéricas y el pipeline de las variables categóricas a las variables categóricas. De esta forma, estaremos paralelizando las transformaciones de las variables númericas y las categóricas, a pesar de que las transformaciones para un mismo tipo de variable sea secuencial.
   Para ver un ejemplo, voy a crear una nueva variable categórica y ver cómo se haría este proceso:

```
from sklearn.compose import ColumnTransformer

X_train2 = X_train.copy()

# Create new categorical column
options = dict({0:'Moscatel',1:'Sultanina',2:'Merlot'})
X_train2['grape_type'] = np.random.randint(0,3, size = X_train.shape[0])
X_train2['grape_type'] = [options.get(grape) for grape in X_train2['grape_type']]

# Create pipeline for numerical variables
numeric_pipe = Pipeline([
   ('scaler', preprocessing.StandardScaler()),
   ('selector', VarianceThreshold())
   ])

# Create pipeline for categorical variable
categorical_pipe = Pipeline([
   ('encoder', preprocessing.OneHotEncoder(drop = 'first'))
   ])

# Create ColumnTransform
col_transf = ColumnTransformer([
   ('numeric', numeric_pipe, X_train2._get_numeric_data().columns.tolist()),
   ('categoric', categorical_pipe, ['grape_type'])
   ])

col_transf_fit = col_transf.fit(X_train2)
X_train2_transf = col_transf_fit.transform(X_train2)
```

```
print('##-- Row 1 Before Transformation --## ')
print(X_train2.iloc[0,:].tolist())
print('##-- Row 1 After Transformation --## ')
print(X_train2_transf[0].tolist())
```

```
##-- Row 1 Before Transformation --##
[13.88, 5.04, 2.23, 20.0, 80.0, 0.98, 0.34, 0.4, 0.68, 4.9, 0.58, 1.33, 415.0, 'Merlot']

##-- Row 1 After Transformation --##
[1.1253192595117543, 2.845764061669694, -0.45902297004005566, 0.2279555555839831, -1.3787844017876945, -2.2193953800197064, -1.7894974173749854, 0.2644746993271315, -1.4437479611809765, 0.05691646573974076, -1.69107399616703, -2.0510335121401804, -0.9613061482603757, 0.0, 0.0]
```

Como ves, hemos aplicado todas las transformaciones a nuestros datos, tanto categóricos como numéricos, de una forma muy sencilla y, además, paralelizada. Además, al igual que el pipeline, puedes guardar el ColumnTransformer para poder aplicar estas mismas transformaciones a los nuevos datos que vayan entrando.