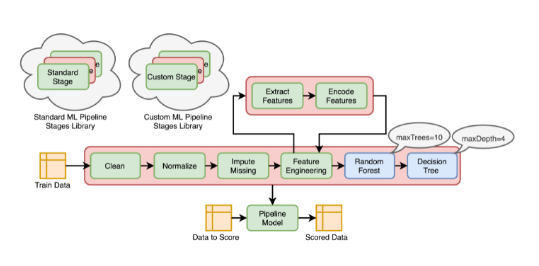
**Pipelines**

Es una **serie de pasos automatizados para transformar los datos** asegurando la **validez y consistencia de los mismos**. Cada paso se alimenta del paso previo. Son **reutilizables**, garantizando así la consistencia de la información. Al agrupar operaciones, proveen un **mayor nivel de abstracción**.

**Al trabajar con datos**, es **muy importante** la **preparación de los mismos**: **limpiarlos**, hacer los **cambios de escala** que sean necesarios, **seleccionar las features**. Esto puede entenderse como un **encadenamiento de procesos** en el cual se pueden incluir también la **aplicación y evaluación de modelos**.



**Pipeline** es una **clase de scikit-learn**. Su **uso más común** es facilitar el **encadenamiento de pasos del preprocessing** junto con un modelo. **Pipeline** es un **estimador** que cuenta con los **métodos fit, predict** y **score**, al igual que otros modelos. **Todos los pasos encadenados** en un pipeline, **excepto el último**, tienen los métodos **fit** y **transform** para **generar una nueva representación** del dato que **se utilizará en el paso siguiente**.

Esquema general para Implementar Pipeline:

1. **Construir una lista de los pasos a ejecutar** en secuencia. Cada paso es una tupla con un **nombre** (a elección) y la instancia de un **estimador**. Es importante haber importado las librerías de los métodos que vamos a llamar. IE: un estimador para escalar datos y otro para clasificarlos.

**En Python:**

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

from sklearn.pipeline import Pipeline

pasos = [(‘scaler’,StandardScaler()), (‘knn’, KNeighborsClassifier())]

1. **Instanciar Pipeline** con los pasos definidos previamente. Vamos a trabajar con el DataSet de Vinos.

pipe = Pipeline(pasos)

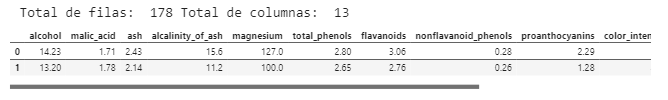
from sklearn import load\_wine

X, y = load\_wine(as\_frame = True, return\_X\_y=True)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size = 0.3, random\_state = 30, stratify = y)

print(‘Totl de filas: ’, X.shape[0], ‘total de columnas, ’, X.shape[1])

X.head(2)



1. **Entrenar la instancia de Pipeline** con los datos de entrenamiento, usando el método fit. Se va a ejecutar para estos datos el paso scaler aplicando fit para aprender la media y desvío de la variable. Luego con transform modifica la escala. Se entrena el modelo con los datos escalados en el paso knn.

pipe.fit(X\_train, y\_train)



1. **Evaluar el modelo generado**. Se usa el método **predict** para evaluar contra los datos de test. Con el método **score** se obtiene el accuracy del modelo. Va a transformar los datos de test con el método transform del Scaler y con ellos va a alimentar el estimador knn, que aplica el método score.

y\_pred = pipe.predict(X\_test)

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

plt.figure(figsize = (5.2))

sns.heatmap(conf\_matrix, annot = True, fmt = ‘d’)

plt.title(‘Confusion matrix’)

plt.ylabel(‘True class’)

plt.xlabel(‘Predicted class’)

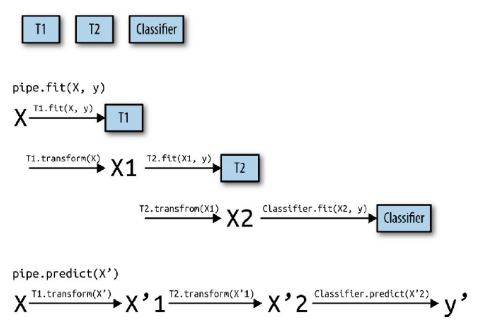
plt.show()



pipe.score(X\_test, y\_test)



Siendo T1 y T2 pasos de preprocesamiento y Classifier un clasificador, acá tenemos un ejemplo gráfico de cómo trabaja Pïpelines.



Con la clase **make\_pipeline** es posible crear un Pipeline al que se le pone un nombre a cada paso automáticamente.

**En Python:**

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

from sklearn.pipeline import Pipeline

pasos = [(‘scaler’, StandardScaler()), (‘knn’, KNeighborsClassigier())]

pipe = Pipeline(pasos)

from sklearn import make\_pipeline

makepipe = make\_pipeline(StandardScaler(), KNeighborsClassifier())

# Con **.steps** podemos ver qué nombre le puso a cada paso.

makepipe.steps



Los estimadores se guardan como lista en el atributo steps:

pasos = [(‘scaler’, StandardScaler()), (‘knn’, KNeighborsClassifier())]

pipe = Pipeline(pasos)

pipe.steps[0]



pipe.steps[1]



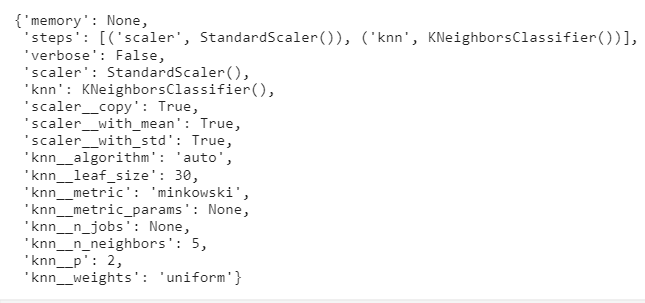
El atributo **named\_steps** guarda la misma información, pero como un diccionario:

pipe.named\_steps[‘scaler’]



Podemos ver los parámetros de los estimadores con el método **get\_params**. Los va a presentar con un formato nombre\_paso\_nombre\_parámetro. Es posible modificarlos con **set\_params.**

pipe.get\_params()



pipe.set\_params(knn\_n\_neighbors = 4)



**Pipeline + GridSearch**

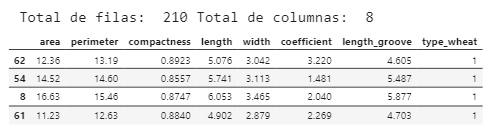
Vamos a trabajar con un DataSet de semillas de trigo en el cual las features son las propiedades geométricas de las semillas y la variable objetivo es el type\_wheat, que es el tipo de semilla 0 – Kama, 1 – Rosa, 2 – Canadian.

**En Python:**

df = pd.read\_csv(‘../Data/sedes\_dataset.csv’)

print(‘Total de filas: ‘, df.shape[0], ‘Total de columnas: ’, df.shape[1])

df.sample(4)

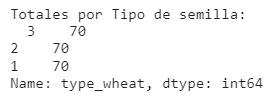


X = df.drop(df[‘type\_wheat’], axis = 1)

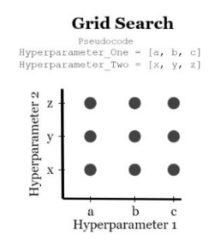
y = df[‘type\_wheat’]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.3, random\_state = 30, stratify = y)

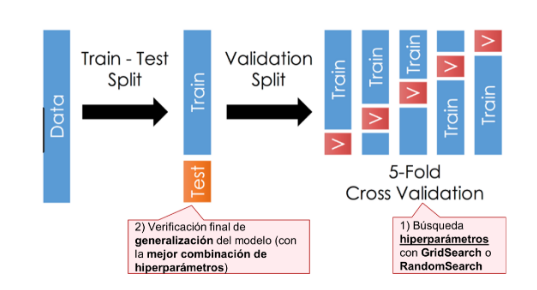
print(‘Totales por tipo de semilla /n ’, pd.value\_counts(df[‘type\_wheat’], sort = True))



**GridSearch** busca la mejor combinación de hiperparámetros dentro de una grilla **Grid** especificada previamente. Se hace la búsqueda de manera exhaustiva para cada valor de la grilla.



Se aplica cada combinación de los hiperparámetros en el dataset de train, se evalúa con cross validation y se registra el score. Una vez que se hizo la prueba con todas las combinaciones posibles, se selecciona la combinación con más alto score, se aplica sobre train y se predice sobre test.



Es posible combinar pipelines con GridSearch:

**En Python:**

pasos = [(‘scaler’, StandardScaler()), (‘knn’, KNeighborsClassifier())]

pipe\_grid = Pipeline(pasos)

# Importamos GridSearchCV y una clase para definir los k fold del esquema de validación.

from sklearn.model import GridSearchCV, StratifiedKFold

folds = StratifiedKFold(n\_splits = 5, shuffle = True, random\_state = 42)

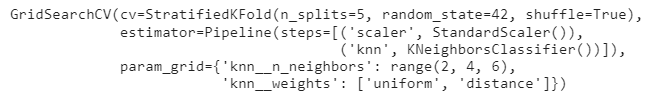
# La Grilla de hiperparámetros debe identificar el paso del pipeline y el hiperparámetros y respectivos valores a evaluar. La sintaxis a usar es nombre\_paso\_nombre\_hiperparámetro.

param\_grid = {‘knn\_k\_neighbors’:range(2,4,6), ‘knn\_\_weigths’: [‘uniform’, ‘distance’]}

# Entonces vamos a combiner los dos métodos instanciando GridSearchCV con la instancia de pipeline, la grilla definida y el método de Cross Validation. Luego pasamos a entrenar el modelo:

grid = GridSearchCV(pipe\_grid, param\_grid, cv = folds)

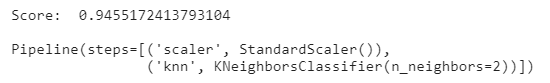
grid.fit(X\_train, y\_train)



#Ahora vamos a ver el score del mejor modelo (**best\_score\_**) y los pasos previos y su configuración (**best\_estimator\_**)**:**

print(‘Score ‘, grid.best\_score\_)

grid.best\_estimator\_



# Ahora vamos a ver la performance con los datos de test:

accuracy\_score(grid.best\_estimator\_.predict(X\_test), y\_test)



**Pipeline + GridSearch + Preprocessing:**

# Algo más que también se puede hacer, además de buscar los mejores hiperparámetros con GridSearch, es explorar la mejor configuración para los pasos previos, tales como chequear si nos conviene escalar con StandardScaler o con MinMaxScaler

# Vamos a generar una lista de pasos que le vamos a pasar a pipeline y lo instanciamos. Sólo definimos como paso el StandardScaler:

pasos = [(‘scaler’, StandardScaler()), (‘knn’, KNeighborsClassifier())]

pipe\_grid\_2 = Pipeline(pasos)

# Importamos GridSearchCV y una clase para definir los kfold del esquema de validación:

from sklearn.model\_selection import GridSearchCV, StratifiedKFold

folds = StratifiedKFold(n\_splits = 5, shuffle = True, random\_state = 42)

# En la grilla de hiperparámetros se incluye el paso ‘scaler’ y sus respectivos valores a evaluar. Es importante haber importado los métodos a evaluar. Si incluimos **None** como opción, lo que estamos indicando es que es factible no escalar, y queremos 3saber qué pasaría si no lo hiciéramos.

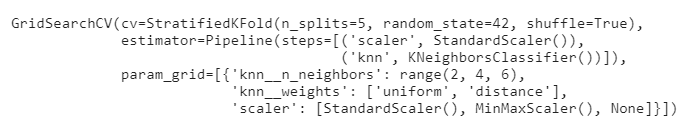
from sklearn.preprocessing import MinMaxScaler

param\_grid\_2 = [{‘scaler’:[StandardScaler(), MinMaxScaler(), None], ‘knn\_\_neighbors’:range(2,4,6), ‘knn\_\_weights’:[‘uniform’, ‘distance’]}]

# Combinamos Pipeline y GridSearchCV y entrenamos el modelo:

grid\_2 = GridSearchCV(pipe\_grid\_2, param\_grid\_2, cv=folds)

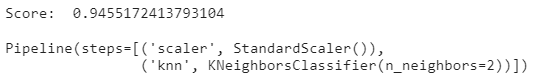
grid\_2.fit(X\_train, y\_train)

****

# Ahora vamos a ver el score del mejor modelo (**best\_score\_**)y los pasos previos y su configuración (**best\_estimator\_**)**.** Para este caso, los valores no mejoraron.

print(‘Score: ’, grid\_2.best\_score\_)

grid\_2.best\_estimator\_

****

# Vemos la performance con los datos de test.

accuracy\_score(grid\_2.best\_estimator\_.predict(X\_test), y\_test)



Con el package **preprocessing** podemos usar funciones y clases que pueden transformar los datos para adecuarlos a nuestros modelos. Si esto no alcanza, nosotros podemos crear nuevas funciones (**Custom Data Transformers**). Vamos a ver dos maneras que nos da scikit-learn para crearlas y usarlas en forma estándar como cualquier otra transformación.

Hasta ahora, venimos aplicando las transformaciones con los métodos **.fit**, **.transform**

y **.fit\_transform**.

**En Python:**

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_train\_scaler = pd.DataFrame(scaler.fit\_transform(X\_train))

X\_test\_scaler = pd.DataFrame(scaler.transform(X\_test))

Una forma es usando la función **FunctionTransformer** de **preprocessing**. Esta función **convierte una función Python** que ya existe en un **Transform**.

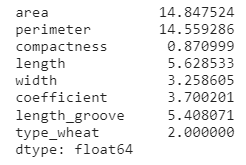
**En Python:**

from sklearn.preprocessing import **FunctionTransformer**

transformer = FunctionTransformer(np.mean)

df\_mean = transformer.transform(df)

df\_mean

****

La otra opción para crear **Custom** DataTransformers es **heredar** los métodos de las clases de scikit-learn para tenerlos disponibles en nuestras propias clases: Con **BaseEstimator** podemos heredar los métodos **get\_params** y **set\_params**. Con **TransformerMixin** podemos heredar los métodos **.fit**, **.transform** y **.fit\_transform**.

**En Python:** Queremos un transforme que multiplique la entrada por un valor determinado.

from skelarn.base import BaseEstimator, TransformerMixin

# Vamos a definir nuestra propia base. Le vamos a pasar las clases **BestEstimator** y **TransformerMixin** para que **herede los métodos** de scikit-learn. Con el **constructor \_\_\_init\_\_\_** definimos los **parámetros que vamos a necesitar**. No necesitamos fit, ya que sólo transformamos. **Con transform indicamos qué vamos a hacer** con los datos.

class FeatureMultiplier(BaseEstimator, TransformerMixin):

def \_\_\_init\_\_\_(self, factor):

self.factor = factor

def fit(self, X, y=None):

return self

def transform(self, X, y = None):

return X \* self.factor

# Ahora vamos a generar una instancia de la clase:

fm = FeeatureMultiplier(2)

# Vamos a tranformar un dato:

test = np.diag((1,2,3))

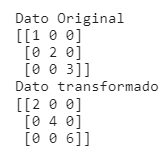
test\_t = fm.transform(test)

print(‘Dato original’)

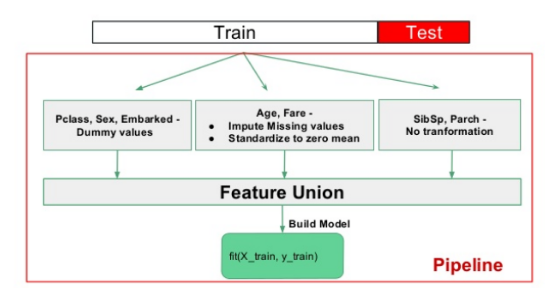
print(test)

print(‘Dato transformado’)

print(test\_t)



Con **featureUnion** podemos combiner varios transformadores y devolver los outputs concatenados. Les aplica a los datos una serie de transformaciones en paralelo y concatena la salida de los mismos en una única matriz. Sirve para combinar mecanismos de extracción de features en un solo transformador. En el siguiente ejemplo, FeatureUnion recibe tres transform para distintos tipos de features, concatena sus salidas y ls entrega como un único dataset al modelo:



Ahora vamos a integrar este concepto de Custom Data Transformers con Pipeline: Vamos a crear dos custom\_data\_transformers: Uno para seleccionar un conjunto de features y otro para devolver una feature discretizada. Vamos a generar dos conjuntos de features a partir de **FeatureUnion** y los transformers. Luego vamos a aplicar GridSearch para evaluar y seleccionar una opción de cada grupo (los dos conjuntos de features; los métodos de procesamiento optando entre StandardScaler(), MinmaxScaler() o None; y los hiperparámetros n\_neighbors o weights.

**En Python:**

from sklearn.pipeline import FeatureUnion, Pipeline

from sklearn.base import BaseEstimator, TransformerMixin

from sklearn.preprocessing import StandardScaler, MinMaxScaler

from sklearn.model\_selection import GridSearchCV, StratifiedKFold

df.head(1)



class FeatureSelection(BaseEstimator, TransformerMixin):

def \_\_init\_\_\_(slef, selected\_features):

self.selected\_features = selected\_features

def fit(self, X, y=None):

return self

def transform(self, X, y=None):

return X[self.selected\_features]

class FeatureDiscretize(BaseEstimator, TransformerMixin):

def \_\_\_init\_\_\_(self, selected\_features):

self.selected\_features = selected\_features

def fit(self, X, y=None):

return self

def transform(self, X, y=None):

return pd.DataFrame(pd.qcut(X[slef.selected\_features], 10, labels = False))

# Vamos a crear los dos conjuntos de features a evaluar a partir de FeatureUnion y los dos Custom\_transformers:

union1 = Featureunion[(‘select’,FeatureSelection(selected\_features = [‘area’, ‘perimeter’])), (‘discret’,FeatureDiscretize() selected\_features = [‘length\_groove])]

union\_2 = Featureunion[(‘select’, FeatureSelection(selected\_features=[‘area’,’perimeter’, ‘length’, ‘width’])), (‘discret’, FeatureDiscretize(selected\_features = ‘compactness’))])

# Ahora generamos la lista de pasos:

pasos = [(‘feature\_engineering’, unión\_1), (‘preprocesamiento’, StandardScaler()), (‘clasificador’, KNeighborsClassifier())]

pipe\_grid\_final = Pipeline(pasos)

# Ahora vamos a ver el score del mejor modelo (**best\_score\_**)ylos pasos previos y su configuración (**best\_estimators\_**).

print(‘Score: ’, )