# Sklearn APIs and Pipelines

Scikit Learn is a popular machine-learning library in Python. It provides a wide range of tools for building machine-learning models. It has a simple API consisting of the following methods:

- `.fit(X, y)`: Fit the model to the data
- `.predict(X)`: Predict the target variable for new data
- `.predict_proba(X)`: Predict the probability of the target variable
- `.score(X, y)`: Evaluate the model on the data
- `.transform(X)`: Transform the data
- `.fit_transform(X, y)`: Fit the model and transform the data

An example of use might for using logistic regression is:

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

A predictor will have the `.predict` method, while a transformer will have the `.transform` method. A transformer can be used to preprocess the data before fitting a model.

Often, models will have attributes ending with an underscore, set after fitting the model. For example, a linear regression model's `coef_` attribute contains the model's coefficients.

## Predictors

A predictor is used to create a label for a new data point. The label can be a class label or a continuous value. The following are some common predictors:

- `LinearRegression`: Linear regression model
- `LogisticRegression`: Logistic regression model
- `DecisionTreeClassifier`: Decision tree classifier
- `RandomForestClassifier`: Random forest classifier
- `KNeighborsClassifier`: K-nearest neighbors classifier

## Hyperparameters

Hyperparameters are the parameters set before the model is trained. They are used to control the behavior of the model. Often„ a hyperparameter will cause a model to be more complex or more flexible. In scikit-learn, there is a convention: increasing the value of hyperparameters starting with `max_` will cause the model to be more complex. For example, increasing the value of `max_depth` in a decision tree will cause the tree to be more complex.

Increasing the value of hyperparameters starting with `min_` will make the model simpler. For example, increasing the value of `min_samples_split` in a decision tree will cause the tree to be more flexible.

## Data Format

Sklearn models work with NumPy and Pandas data structures. I prefer to use Pandas data frames because they are more flexible and easier to understand. Having a column label is invaluable for understanding the data.

The `X` parameter (capitalized because it is a matrix) is a 2D array with the shape `(n_samples, n_features)`. Each row represents a sample (or example). Statisticians call the rows observations.

The columns represent the features of the sample. Statisticians call these columns independent variables, and machine learning engineers call them features.

The `y` parameter is a 1D array with the shape `(n_samples,)`. It has the target for each row in `X`. Statisticians call this the dependent variable. Machine learning engineers call this the target, label, class (in the case of classification), or outcome.

For many models, the data needs to be:

- Numeric
- Without missing values
- Scaled

Most models work with numeric data, but some can with categorical data. For example, decision trees can work with categorical data. The math behind many models also does not support missing values. Missing values can be *imputed* to replace with real values. Scaling the data can improve the performance of many models. Standardized scaling is the most common method, which scales the data with a mean of 0 and a standard deviation of 1.

## Pipelines

Pipelines are used to preprocess the data before fitting a model. Sklearn provides a transformer class to manipulate data. You can chain these operations together using a pipeline. A pipeline is a sequence of transformers followed by a predictor. The pipeline has the same API as a predictor. The following is an example of a pipeline:

Let's look at an example using the Ames Iowa dataset.

```
# import ames data
from sklearn.datasets import fetch_openml
```

```
import numpy as np
import pandas as pd

ames = fetch_openml(name="house_prices", as_frame=True)

ames_df = pd.DataFrame(data=ames['data'], columns=ames['feature_names'])
target = pd.Series(ames['target'])
```

## Transformers

Transformers are used to *transform* 2-dimensional input to 2-dimmensional output. They are used to fill in missing values, standardize data and more.

The important methods are `.fit` which reads the input and teaches the transformer how to do its transformation and `.transform`, which does the actual transformation.

It's imperative that `.fit` is only called on the training data and not on the testing or data you are making predictions for.

```
# median imputation
from sklearn import impute, set_config
set_config(transform_output='pandas')

median_imputer = impute.SimpleImputer(strategy='median')
print(median_imputer.fit_transform(ames_df[['LotFrontage']]))
```

## Pipelines

Pipelines allow chaining sklearn transformers and predictors. They have the same interface as other sklearn objects.

To create a pipeline, use the `Pipeline` class and pass in a sequence of tuples of name, transformer.

Here is a pipeline that imputes missing values with the median and then standardizes the results.

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn import set_config

set_config(transform_output='pandas')

num_pipeline = Pipeline([
    ('num_impute', SimpleImputer(strategy='median')),
    ('num_std', StandardScaler())
])

num_pipeline.fit_transform(ames_df[['LotFrontage']])
```

## Column Transformers

In sklearn, a `ColumnTransformer` is used to apply different transformers to different columns. It is useful when you have a mix of numeric and categorical columns. You can apply one pipeline to the numeric columns and another pipeline to the categorical columns.

```
# import column transformer
from sklearn.compose import ColumnTransformer
# import one hot encoder
from sklearn.preprocessing import OneHotEncoder

num_cols = ['LotFrontage', 'LotArea', 'OverallQual']
num_pipeline = Pipeline([
    ('num_imputer', SimpleImputer(strategy='median')),
    ('num_std', StandardScaler())
])

cat_cols = ['Neighborhood', 'Electrical', 'Utilities']
cat_pipeline = Pipeline([
    ('cat_impute', SimpleImputer(strategy='constant', fill_value='Other')),
    ('cat_ohe', OneHotEncoder(sparse_output=False, max_categories=10))
])

col_transformer = ColumnTransformer([
    ('num_pipeline', num_pipeline, num_cols),
    ('cat_pipeline', cat_pipeline, cat_cols)
], remainder='drop')

col_transformer.fit_transform(ames_df)
```

The output of the column transformer is a pandas dataframe. It will insert the pipeline name in front of the column that it transforms.

## Final Pipeline

You can stick a column transformer into another pipeline if needed. Here is a PCA pipeline:

```
from sklearn.compose import ColumnTransformer
from sklearn.decomposition import PCA
from sklearn.preprocessing import OneHotEncoder

num_cols = ['LotFrontage', 'LotArea', 'OverallQual']
num_pipeline = Pipeline([
    ('num_imputer', SimpleImputer(strategy='median')),
    ('num_std', StandardScaler())
```

```
])

cat_cols = ['Neighborhood', 'Electrical', 'Utilities']
cat_pipeline = Pipeline([
    ('cat_impute', SimpleImputer(strategy='constant', fill_value='Other')),
    ('cat_ohe', OneHotEncoder(sparse_output=False, max_categories=10))
])

col_transformer = ColumnTransformer([
    ('num_pipeline', num_pipeline, num_cols),
    ('cat_pipeline', cat_pipeline, cat_cols)
], remainder='drop')

pca_pipeline = Pipeline([
    ('preprocess', col_transformer),
    ('pca', PCA())
]
)

pca_pipeline.fit_transform(ames_df)
```

To pull off attributes from the items in the pipeline, access the named attribute from `named_steps`

```
pca = pca_pipeline.named_steps['pca']

pca.components_
```

## Splitting Data

An important thing to remember is to split the data into training and testing data. This is done using the `train_test_split` function. If you are doing classification, remember to use the `stratify` parameter so that the training and testing sets have the same proportion of classes.

To get similar ratios while doing regression, you can use the `stratify` parameter with the `pd.qcut` function.

```
from sklearn.model_selection import train_test_split

X = ames_df
y = target

# split with qcut
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42,
                                                    stratify=pd.qcut(y, q=4, labels=False))
```

## Linear Regression Pipeline

```
from sklearn.compose import ColumnTransformer
from sklearn.decomposition import PCA
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder

num_cols = ['LotFrontage', 'LotArea', 'OverallQual']
num_pipeline = Pipeline([
    ('num_imputer', SimpleImputer(strategy='median')),
    ('num_std', StandardScaler())
])

cat_cols = ['Neighborhood', 'Electrical', 'Utilities']
cat_pipeline = Pipeline([
    ('cat_impute', SimpleImputer(strategy='constant', fill_value='Other')),
    ('cat_ohe', OneHotEncoder(sparse_output=False, max_categories=10, handle_unknown='ignore'))
])

col_transformer = ColumnTransformer([
    ('num_pipeline', num_pipeline, num_cols),
    ('cat_pipeline', cat_pipeline, cat_cols)
], remainder='drop')

lr_pipeline = Pipeline([
    ('preprocess', col_transformer),
    ('lr', LinearRegression())
]
)

lr_pipeline.fit(X_train, y_train)


>>> print(lr_pipeline.score(X_test, y_test))
0.6515422044770535
```

## XGBoost Model

```
import xgboost as xgb
from sklearn.base import BaseEstimator, TransformerMixin

class CatTransformer(BaseEstimator, TransformerMixin):
```

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

    def transform(self, X):
        return (X
                .assign(**X.select_dtypes(object).astype('category'))
        )

class ColumnSelector(BaseEstimator, TransformerMixin):
    def __init__(self, cols_to_keep):
        self.cols_to_keep = cols_to_keep

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

    def transform(self, X):
        return X[self.cols_to_keep]

xgb_pipeline = Pipeline([
    ('keeper', ColumnSelector([*num_cols, *cat_cols])),
    ('preprocess', CatTransformer()),
    ('xgb', xgb.XGBRegressor(enable_categorical=True))
])

xgb_pipeline.fit(X_train, y_train)


>>> print(xgb_pipeline.score(X_test, y_test))
0.6008727923708206
```