Polynomial Regression

Estimated time needed: **40** minutes

What if your data is more complex than a straight line? Surprisingly, you can use a linear model to fit nonlinear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called Polynomial Regression.

There are two factors when determining model performance: overfitting and underfitting. Overfitting is when the model is too complex and does well on the training data but not on the test data. Underfitting is when the model is too simple and performs poorly on the training and testing data sets.

Overfitting is simple to deal with, using methods like regularization, which we will discuss in the next lab. To deal with underfitting, we can build a more complex model using methods like polynomial regression. If making a more complex model does not work, this may involve using more data to train the model on or obtaining new features. As this process is complex, it's better to determine if the model can overfit the data first. Therefore, in this section, we will use Polynomial Regression to overfit the data to determine if we have an adequate amount of data.

In this notebook, we will explore Polynomial Regression and perform polynomial transform using individual features as well as multiple features.

Objectives

After completing this lab you will be able to:

- Understand the concept of overfitting versus underfitting
- Apply polynomial transforms to data
- Perform hyperparameters grid search on a model, using validation data

```
Comienza a programar o generar con IA.
```

Setup

For this lab, we will be using the following libraries:

- `pandas` for managing the data.
- `numpy` for mathematical operations.
- `seaborn` for visualizing the data.
- `matplotlib` for visualizing the data.
- `sklearn` for machine learning and machine-learning-pipeline related functions.
- `scipy` for statistical computations.

Import the required libraries

The following required modules are pre-installed in the Skills Network Labs environment. However, if you run this notebook commands in a different Jupyter environment (e.g. Watson Studio or Ananconda), you will need to install these libraries by removing the `#` sign before `!mamba` in the code cell below.

```
#!pip install -U scikit-learn
!pip install pandas
!pip install numpy
!pip install seaborn
!pip install matplotlib
!pip install scikit-learn
```

```
# Surpress warnings:
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

```
import pandas as pd
import numpy as np

import seaborn as sns
import matplotlib.pylab as plt
%matplotlib inline

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import scale
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
```

The function below will calculate the R^2 on each feature given the a input model.

```
def get_R2_features(model,test=True):
    #X: global
    features=list(X)
    features.remove("three")

    R_2_train=[]
    R_2_test=[]

    for feature in features:
        model.fit(X_train[[feature]],y_train)

        R_2_test.append(model.score(X_test[[feature]],y_test))
        R_2_train.append(model.score(X_train[[feature]],y_train))

    plt.bar(features,R_2_train,label="Train")
    plt.bar(features,R_2_test,label="Test")
    plt.xticks(rotation=90)
    plt.ylabel("$R^2$")
    plt.legend()
    plt.show()
    print("Training R^2 mean value {} Testing R^2 mean value {} ".format(str(np.mean(R_
    print("Training R^2 max value {} Testing R^2 max value {} ".format(str(np.max(R_2_t
```

The function below will plot the distribution of two inputs.

```
def  plot_dis(y,yhat):

    plt.figure()
    ax1 = sns.distplot(y, hist=False, color="r", label="Actual Value")
    sns.distplot(yhat, hist=False, color="b", label="Fitted Values" , ax=ax1)
    plt.legend()

    plt.title('Actual vs Fitted Values')
    plt.xlabel('Price (in dollars)')
    plt.ylabel('Proportion of Cars')

    plt.show()
    plt.close()
```

Reading and understanding our data

Reading and understanding our data

For this lab, we will be using the car sales dataset, hosted on IBM Cloud object storage. This dataset can also be found and downloaded from kaggle.com, an open public data source. The dataset contains all the information about cars, a name of a manufacturer, all car's technical parameters and a sale price of a car.

This dataset has already been pre-cleaned and encoded (using one-hot and label encoders) in the Linear Regression Notebook.

Let's read the data into *pandas* data frame and look at the first 5 rows using the `head()` method.

```
data = pd.read_csv('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.clouc
data.head()
```

We can find more information about the features and types using the `info()` method.

```
data.info()
```

We have 35 features in our dataset after the one hot encoding.

Before we begin our polynomial analysis, let's visualize some of the relationships between our features and the target variable, 'price'.

```
sns.lmplot(x = 'curbweight', y = 'price', data = data, order=2)
```

```
sns.lmplot(x = 'carlength', y = 'price', data = data, order=2)
```

The relationship is more curved.

Exercise 1

In this Exercise, visualize the relationship between the 'horsepower' and the target variable, 'price'.

```
sns.lmplot(x = 'horsepower', y = 'price', data = data, order=2)
```

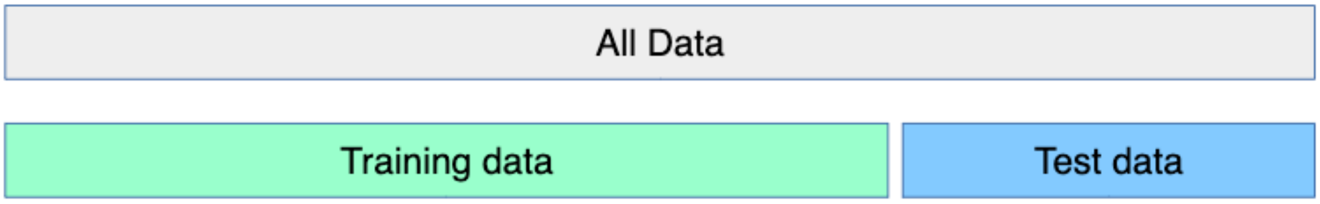
Data Preparation

Let's first split our data into `X` features and `y` target.

```
X = data.drop('price', axis=1)
y = data.price
```

Train Test Split

Now that we have split our data into training and testing sets, the training data is used for your model to recognize patterns using some criteria,the test data set it used to evaluate your model, as shown in the following image:



source scikit-learn.org

Now, we split our data, using `train_test_split` function, into the training and testing sets, allocating 30% of the data for testing.

```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3, random_state=4
print("Number of test samples:", X_test.shape[0])
print("Number of training samples:", X_train.shape[0])
```

Multiple Features

Let's create a `LinearRegression` object, called `lm`.

```
lm = LinearRegression()
```

Now, let's fit the model with multiple features on our X_train and y_train data.

```
lm.fit(X_train, y_train)
```

We apply `predict()` function on the testing data set.

```
predicted = lm.predict(X_test)
```

Let's calculate the `r2_score()` on both, training and testing data sets.

```
print("R^2 on training  data ",lm.score(X_train, y_train))
print("R^2 on testing data ",lm.score(X_test,y_test))
```

We can plot distributions of the predicted values versus the actual values.

```
plot_dis(y_test,predicted)
```

Below, we will view the estimated coefficients for the linear regression problem.

```
{col:coef for col,coef in zip(X.columns, lm.coef_)}
```

As we see, the first two coefficients are too large to plot, so we'll drop them and plot the rest of the coefficients.

```
plt.bar(X.columns[2:],abs(lm.coef_[2:]))
plt.xticks(rotation=90)
plt.ylabel("$coefficients$")
plt.show()
```

Usually, we can interpret the lager coefficients as having more importance on the prediction, but this is not always the case, so let's look at the individual features.

Individual Features

We can train the model and plot our R^2 for each of the features on the training and testing data sets, using the function `get_R2_features`.

```
get_R2_features(lm)
```

```
get_R2_features(lm)
```

From the above plot, we see that some individual features perform similarly to using all the features (we removed the feature `three`), in addition, we see that smaller coefficients seem to correspond to a larger R^2, therefore larger coefficients correspond to overfiting.

Exercise 2

In this Exercise, calculate the R^2 using the object Pipeline for Linear Regression and apply `StandardScaler()` to all features, then use the function `plot_dis` to compare the predicted values versus the actual values.

```
pipe = Pipeline([('ss',StandardScaler() ),('lr', LinearRegression())])
pipe.fit(X_train,y_train)
print("R^2 on training  data ", pipe.score(X_train, y_train))
print("R^2 on testing data ", pipe.score(X_test,y_test))
predicted = pipe.predict(X_test)
plot_dis(y_test,predicted)
```

Exercise 3

In this Exercise, calculate the R^2 using the object Pipeline with `StandardScaler()` for each individual features using the function `get_R2_features` .

```
pipe = Pipeline([('ss',StandardScaler() ),('lr', LinearRegression())])
get_R2_features(pipe)
```

Polynomial Features

Multiple Features

Polynomial transform is a simple way to increase the complexity of the model, but we must be mindful of overfilling. Below, we will perform a second degree (degree=2) polynomial transformation.

```
poly_features = PolynomialFeatures(degree=2, include_bias=False)
```

Now, we transform the training and testing data sets.

```
X_train_poly = poly_features.fit_transform(X_train)
X_test_poly = poly_features.transform(X_test)
```

`X_train_poly` and `X_test_poly` now contain the original features of X plus the square of these features and the cross-terms combination. Let's check the shape of the newly created train and test sets.

```
print(X_train_poly.shape)
```

```
print(X_test_poly.shape)
```

Altogether, we have 665 features. Now, we fit the model with the newly created features.

```
lm = LinearRegression()
lm.fit(X_train_poly, y_train)
```

And we make predictions.

```
predicted = lm.predict(X_train_poly)
```

Again, we can ckeck the `r2_score()` on both, training and testing data sets.

```
print("R^2 on training data:", lm.score(X_train_poly, y_train))
print("R^2 on testing data:", lm.score(X_test_poly,y_test))
```

We see the model has a negative R^2 on the test data set, this is sign of overfiting.

Individual Features

Data Pipelines simplify the steps of processing the data. We use the module `Pipeline` to create a pipeline. We also use `PolynomialFeatures` as a step in our pipeline.

```
Input=[ ('polynomial', PolynomialFeatures(include_bias=False,degree=2)), ('model', Lir
```

We can repeat the steps above, using the `Pipleine` object.

```
pipe=Pipeline(Input)
pipe.fit(X_train, y_train)
```

We can see the results are identical.

```
print("R^2 on training  data:", pipe.score(X_train, y_train))
print("R^2 on testing data:", pipe.score(X_test,y_test))
```

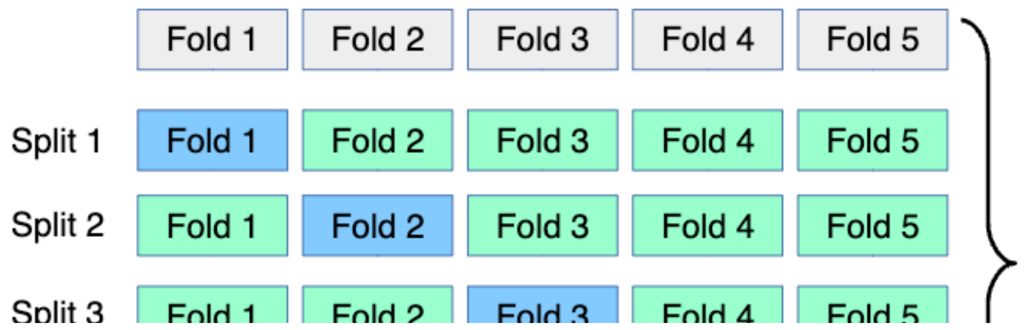
We can train our model on each of the features using the Polynomial Feature transform of the second degree. Then we can plot our R^2.

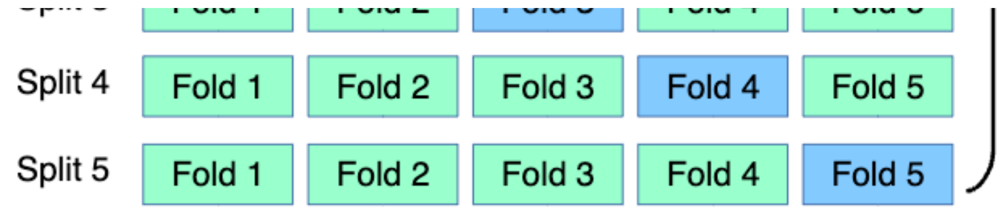
```
get_R2_features(pipe)
```

Feature with the max R^2 is higher than when using all the features.

GridSearch and Pipeline

In this section of the notebook, we will define a pipeline object, then use GridSearch to find the best hyper-parameters of the model by using cross-validation method of the parameter grid, as shown in the figure below. A 5-fold cross validation is used by default. We will learn more about k-fold cross validation in the next, Cross Validation lesson of the Course.





source scikit-learn.org

We create `PolynomialFeatures()` pipeline.

```
Input=[ ('scaler', StandardScaler()), ('polynomial', PolynomialFeatures(include_bias=F
pipe=Pipeline(Input)
```

To search for the best combination of hyperparameters, we create a **GridSearchCV** object with dictionary of parameter values. The parameters of pipelines can be set using the name of the key separated by '__', then the parameter name of the key.

```
param_grid = {
    "polynomial__degree": [1, 2, 3]


}
```

polynomial__degree: is the degree of the polynomial.

```
search = GridSearchCV(pipe, param_grid, n_jobs=1)
```

```
pipe.fit(X_train, y_train)
```

```
search.fit(X_test, y_test)
```

The object finds the best parameter values on the validation data. We can obtain the estimator with the best parameters and assign it to the variable `best`, as follows:

```
best=search.best_estimator_
best
```

We see the degree is one and normalize is `Ture` .

Now, we test our model on the test data to see the best score.

```
best.score(X_test,y_test)
```

Finally, we can plot a distribution of the predicted values versus the actual values.

```
predicted=best.predict(X_test)
plot_dis(y_test,predicted)
```

As we see, the result is almost perfect!

Exercise 4

In this Exercise, calculate the R^2 using the object `Pipeline` with `GridSearch` for each individual features.

```
features=list(X)
```

```
R_2_train=[]
R_2_test=[]

for feature in features:
    param_grid = {
    "polynomial__degree": [ 1, 2,3,4,5],
    "model__positive":[True, False]}
    Input=[ ('polynomial', PolynomialFeatures(include_bias=False,degree=2)), ('model',
    pipe=Pipeline(Input)
    print(feature)
    search = GridSearchCV(pipe, param_grid, n_jobs=2)
    search.fit(X_test[[feature]], y_test)
    best=search.best_estimator_

    R_2_test.append(best.score(X_test[[feature]],y_test))
    R_2_train.append(best.score(X_train[[feature]],y_train))
```

```
plt.bar(features,R_2_train,label="Train")
plt.bar(features,R_2_test,label="Test")
plt.xticks(rotation=90)
plt.ylabel("$R^2$")
```

```
    plt.legend()
    plt.show()
    print("Training R^2 mean value {} Testing R^2 mean value {} ".format(str(np.mean(R_2_t
    print("Training R^2 max value {} Testing R^2 max value {} ".format(str(np.max(R_2_trai
```