IBM Data Science Professional Certificate

Course 7: Data Analysis with Python

Week 1: Importing Datasets

Why Data Analysis?

- data is everywhere.
- data analysis & data science help us answer questions from data.
- data analysis plays an important role in:
  - discovering useful information
  - answering questions
  - predicting the future or the unknown

Python Packages for Data Science

- Scientific Computing Libraries
  - Pandas
    - Data structures & tools (dataframes)
    - Designed to provide easy indexing functionality
    - Offers tools and data structure for effective data manipulation and analysis
  - Numpy
    - Arrays & matrices
  - SciPy
    - Integrals, solving differential equations, optimization
    - Helpful in some data visualization.
- Data Visualization Libraries
  - Matplotlib
    - Plots & graphs, most popular visualization library
  - Seaborn
    - Plots: heat maps, time series, violin plots
    - Based on matplotlib
- Algorithmic Libraries
  - SciKit-Learn
    - Machine learning: statistical modeling, regression, classification, clustering, etc
    - Built on Numpy, Scipy, and Matplotlib.
  - Statsmodels
    - Explore data, estimate statistical models, perform statistical tests

Importing Data in Python

Importing:

- the process of loading and reading data into Python from various resources

- Two important properties include:
  - format
    - .csv, .json, .xlsx, .hdf, etc
  - file path of dataset
    - computer: /desktop/mydata.csv
    - internet: https//website.com

Importing process:

1. import pandas library
2. store the location of the dataset in a variable
3. use the appropriate python pandas.read function on the dataset to create a dataframe
4. view the dataframe to double check your work
   - df
   - df.head()
   - df.tail()

Adding headers:

- replace default headers with (df.columns = headers), where headers is a variable containing the list of column headers you want.

NOTE: pandas.read_csv assumes there is a header line in the file. You can set header = None if there isn't one.

Exporting Data in Python

Exporting:

- Preserve progress at any time by saving a modified dataset
- Use the appropriate df.to_ method to export to a specific path location.

Export process:

1. specify the file path, which includes the file name that you want to write to
2. use the df.to_csv(path) method

Data Format	Read	Save
csv	pd.read_csv()	df.to_csv()
json	pd.read_json()	df.to_json()
Excel	pd.read_excel()	df.to_excel()
sql	pd.read_sql()	df.to_sql()
hdf	pd.read_hdf()	df.to_hdf()

Basic insights from the data:

- understand your data before you begin any analysis

- check for:
  - data types
    - check for info and potential type mismatches
    - check for compatability with python methods
    - use df.dtypes to check data types
  - data distribution
    - return a statistical summary with df.describe()
      - skips non-numerical rows automatically, but you can include = 'all' to get summaries for them all
    - return a concise summary of the dataframe with df.info()
- locate potential issues with the data

Accessing Databases with Python

Recall the interaction:

- user, jupyter / python programs, API calls, DBMS

Concepts of the Python DB API:

- Connection Objects
  - database connections
  - manage transactions
- Cursor Objects
  - database queries
  - Methods
    - cursor()
      - returns a new cursor object using the connection
    - commit()
      - commit any pending transactions to the database
    - rollback()
      - causes the database to rollback to the start of any pending transaction
    - close()
      - closes the database connection

Example database connection

from dbmodule import connect

create connection object

connection = connect('databasename', 'username', 'password')

create cursor object

cursor = connection.cursor()

run queries

```
cursor.execute('select * from mytable')

results = cursor.fetchall()
```

free resources

```
cursor.close()

connection.close()
```

Week 2: Data Wrangling

Data Pre-Processing

Pre-processing:

- The process of converting or mapping data from the initial 'raw' form into another format in order to prepare the data for further analysis.
- AKA: data wrangling, data cleaning
- This could include:
  - handling missing values
  - formatting data
  - normalizing data
  - binning data

Missing Values

Missing Values:

- occur when no data value is stored for a variable in an observation
- could be represented as ?, N/A, 0, or just a blank cell.

How to deal with missing data:

- check with the data collection source to see if you can find the value
- drop the missing values (whichever option has the least amount of impact)
  - drop the variable
  - drop the data entry
  - df.dropna()
    - axis = 0 drops an entire row
    - axis = 1 drops the entire column
    - to modify the dataframe, set inplace = True
- replace the missing values
  - replace with an average of similar datapoints
  - replace it by frequency (mode: the most common value in that column)
  - replace it based on other functions
  - df.replace(missing_value, new_value)
- leave it as missing data

Data Formatting

Data Formatting:

- Data is usually collected from different places and stored in different formats
- Bringing data into a common standard of expression allows users to make meaningful comparisons.
- You can update entire columns with calculations.
- Update incorrect data types.
  - df.dypes() to identify data type
  - df.astype() to convert data type

Data Normalization

Data Normalization:

- Make different variables that have different ranges more uniform and consistent.
- Normalization enables a fair comparison between different featuers, making sure that they have the same impact on analysis.

Approaches for normalization

- Simple Feature Scaling
  - ${x}_{new} = {x}_{old}/{x}_{max}$
  - This divides each value by the maximum value of that variable, which makes the new values range between 0 and 1.
  - df['column'] = df['column']/df[column].max()
- Min-Max
  - ${x}_{new} = ({x}_{old} - {x}_{min}) /({x}_{max} - {x}_{min})$
  - This results in the new values ranging between 0 and 1.
  - df['col'] = (df['col'] - df[col].min()) / (df[col].max() - df[col].min())
- Z-Score
  - ${x}_{new} = ({x}_{old} - \mu) / \sigma$
  - The resulting values hover around zero and typically range between [-3, 3].
  - df['col'] = (df['col'] - df['col'].mean())/df['col'].std()

Binning in Python

Binning:

- the grouping of values into 'bins'
- converts numerical into categorical variables
- group a set of numerical values into a set of bins
  - use bins = np.linspace(min, max, number_of_bins) to return the bins over the specified min, max interval.

    - name_variable = [name1, name2, name3...]
    - use pd.cut(df[column], bins, labels = name_variable, include lowest=True) to segment and sort the data values into bins.
- Once divided into bins, you can create histograms of the data.

Turning Categorical into Quantitative variables

Categorical Variables:

- problem: most stats models cannot take in objects or strings as input.
- Solution: add dummy variables for each unique category and assign numerical values in each category.
    - "one-hot encoding"
- use pd.get_dummies() method to convert categorical variables to dummy variables (0 or 1)

Week 3: Exploratory Data Analysis

Exploratory Data analysis (EDA):

- Preliminary step in data analysis to:
    - summarize main characteristics of the data
    - gain better understanding of the data set
    - uncover relationships between variables
    - extract important variables
- What characteristics have the biggest impact on the question we want to solve / data we want to understand?

Useful Tools:

- Descriptive Statistics: describe basic features of the data set and obtain a short summary about the sample and measures of the data.
- GroupBy: grouping of data and how it can help transform the data
- Correlation between variables
- Statistical correlation methods
    - pearson correlation
    - correlation heat maps

Descriptive Statistics

Descriptive Statistics:

- Describe basic features of data
- Give short summaries about the sample and measure of the data

Useful methods for descriptive statistics:

- df.describe()
    - computes basic statistics for all numeric variables in the dataframe

    - mean, total number of data points, standard deviation, quartiles, and extreme values.
    - NaN values are skipped

- df.value_counts()
  - can be used to summarize categorical data by dividing the data into discrete groups to be counted

- box plots: sns.boxplot(x = 'xrange', y = 'yrange', data = df)
  - visualize the median, quartiles, extremes, and outliers

- scatter plots: plt.scatter(x = df['col1'], y = df['col2'])
  - each observation is represented as a point
  - show the relationship between two variables
    - predictor/independent variable (x-axis) is used to predict an outcome
    - target/dependent variable (y-axis) is the variable you are trying to predict

GroupBy in Python

Grouping data:

- use Pandas df.Groupby() method:
  - can be applied to categorical variables
  - group data into categories
  - can be used on single or multiple variables

Pivot tables:

- Use Pandas method df.Pivot() method:
  - one variable is displayed along the columns and the other variable is displayed along the rows
  - Helps make grouped data easier to understand and visualize

Heatmaps:

- Use plt.pcolor(df_pivot, cmap = 'RdBu') method:
  - Plot target variables over multiple variables

Correlation

Correlation:

- a statistical metric for measure to what extent different variables are interdependent.
  - Correlation does not imply causation.
- Correlation can be positive or negative.
  - Correlation can be weak (almost flat, graphically) or strong (steep slope, graphically)
- You can visualize the correlation between two variables using a scatter plot with a linear regression line, which indicates the relationship between the two.

- sns.regplot(x = , y = , data = df)

Pearson Correlation

Correlation Statistical Methods for numeric variables:

- Pearson Correlation
  - use SciPy's stats package to calculate
    - pearson_coef, p_value = stats.pearsonr(df['col'], df['col2'])
  - one way to measure the strength of the correlation between continuous numerical variables.
  - gives two values:
    - correlation coefficient
      - a value close to 1 implies strong positive correlation
      - a value close to -1 implies strong negative correlation
    - P-value
      - tells us how certain we are about the correlation that we calculated.
      - a value less than .001 gives strong certainty about the correlation
      - a value between .001 and .05 gives moderate certainty
      - a value between .05 and .1 gives weak certainty
      - a value larger than .1 gives no certainty at all
  - you can say that there is a strong correlation when the correlation coefficient is close to +/-1 and the p-value is less than .001.

Chi-Square Test

Assocation between two categorical variables:

- Chi-square test for Association
  - intended to test how likely it is that an observed distribution is due to chance.
  - it measures how well the observed distribution of data fits with the distribution that is expected if the variables are independent
  - it tests a null hypothesis that the variables are independent.
  - the test then compares the observed data to the values that the model expects if the data was distributed in different categories by chance.
  - Any time the observed data doesn't fit within the model of the expected values, the probability that the variables are dependent becomes stronger, thus proving the null hypothesis incorrect.
  - This test does not tell you the type of relationship that exists, only that it exists.
  - you can first find the observed counts of the variable(s) using a crosstab from the pandas library (aka, a contingency table if the relationship is between two categorical variables)
    - sum((observed value - expected value)^2 /expected value
    - expected value: (row total * column total) / grand total
    - degree of freedom = (row-1)*(column-1)
    - if the resulting value in the chi-square table is <.05, you can reject the null hypothesis that the variables are independent and conclude there is a correlation.

- In python, you can use the chi square contingency function in the scipy.stats package.
  - scipy.stats.chi2_contingency(cont_table, correction = True)
    - results in [chi-square test value, p-value, degree of freedom] as well as an array with the values of each expected value in the contingency table

Week 4: Model Development

Model Development

Model:

- A model can be thought of as a mathematical equation used to predict a value given one or more other values.
- It relates one or more independent variables to dependent variables.
- The more relevant data you have, the more accurate your model is usually.

Linear Regression

Simple Linear Regression (SLR):

- Linear regression refers to one independent variable to make a prediction.
- A method to help understand the relationship between two variables, the predictor/independent variable x and target/dependent variable y.
- We want to create a linear relationship between the below variables.
  - ${y} = {b}_{0} + {b}_{1}{x}$
    - ${b}_{0}$: the intercept
    - ${b}_{1}$: the slope
- Fit: we take several training points of the relationship to train the model. The results of the training points are the parameters, usually stored into a dataframe or numpy array.
- Noise: In most cases, many factors influence these variables. This uncertainty is taken into account by assuming a small random value is added to the point on the line. This is called noise.
- Once you have the model set up and running $\hat{y}$ is used to indicate that this is an estimate/prediction of the value of y. We can then use this model to predict values we haven't yet seen.

Linear Regression Model in Python:

1. import linear_model from scikit-learn
   - from sklearn.linear_model import LinearRegression
2. create linear regression object using the constructor:
   - lm = LinearRegression()
3. define the predictor and target variables
   - x = df[['col']]
   - y = df[col]
4. use lm.fit to fit the model, i.e. find the parameters.

- lm.fit(x,y)

5. obtain a prediction in an array
   - yhat = lm.predict(x)
6. view the intercept:
   - lm.intercept_
7. view the slope:
   - lm.coef_
8. The relationship between the variables is given by:
   - ${y} = {b}_{0} + {b}_{1}{x}$

Define the response variable (y) as the focus of the experiment and the explanatory variable (x) as a variable used to explain the change of the response variable.

Multiple Linear Regression:

- Multiple linear regression refers to multiple independent variables to make a prediction
- used to explain the relationship between one continuous target (y) variable and two+ predictor (x) variables
  - for example, the linear relationship of four predictor variables would look like:
    - ${\hat{y}} = {b}_{0} + {b}_{1}{x}_{1}+ {b}_{2}{x}_{2}+ {b}_{3}{x}_{3} + {b}_{4}{x}_{4}$

Multiple Linear Regression in Python:

1. import linear_model from scikit-learn
   - from sklearn.linear_model import LinearRegression
2. create linear regression object using the constructor:
   - lm = LinearRegression()
3. Extract the predictor variables and store them into a variable
   - z = df[['col1', 'col2', 'col3', 'col4']]
4. train the model
   - lm.fit(z, df['ycolumn'])
5. obtain a prediction
   - yhat = lm.predict(x)
6. find intercept
   - lm.intercept_
7. find coefficients
   - lm.coef_
8. Find the relationship

NOTE: it is helpful to replace the dependent variable names with actual names to keep their values readable.

Model Evaluation using Visualization

Regression Plot:

- it gives a good estimate of:

    - the relationship between two variables
    - the strength of the correlation
    - the direction of the relationship (positive, negative)
- The regression plot shows a combination of:
    - the scatterplot, where each point represents a different y
    - the fitted linear regression line ($\hat{y}$)
- How to create:
    - import seaborn as sns
    - sns.regplot(x = independentvalue , y = dependentvalue, data =)
    - plt.ylim(0,)

Residual Plot:

- represents the error between the actual value and the predicted value.
- This difference value is calculated and then plotted on the vertical axis with the independent variable as the horizontal.
- We expect to see that the results have zero mean; i.e., they are distributed evenly around the x axis with similar variance. This indicates that a linear plot is appropriate.
- If the residual plot points aren't evenly spread, it indicates a linear plot is not appropriate and a nonlinear model may fit better.
- How to create:
    - import seaborn as sns
    - sns.residualplot(df[dependentvariable], df[independentvariable])

Distribution Plot:

- Counts the predicted value versus the actual value.
- Very useful for visualizing models with more than one independent variable.
- Count the plot the number of predicted points that are approximately equal to each bin of values.
- Then repeat the process for the number of actual points that are approximately equal to each bin of values.
- Pandas will convert this histogram into a distribution (as histograms are only for discrete values).
- This graphically shows how accurate/inaccurate your model is within certain ranges.
- How to create:
    - import seaborn as sns
    - ax1 = sns.distplot(df['actual values'], hist = False, color = 'r', label = "actual value')
    - sns.distplot(yhat, hist = False, color = 'b', label = "fitted values", ax = ax1

Polynomial Regression and Pipelines

Polynomial Regression:

- a special case of the general linear regression model
- useful for describing curvilinear relationships

    - Curvilinear relationships occur by squaring or setting higher-order terms of the predictor variables
    - Example: Quadratic/2nd order
        - ${\hat{y}} = {b}_{0} + {b}_{1}({x}_{1})^1+ {b}_{2}({x}_{1})^2$
    - Example: cubic / 3rd order
        - ${\hat{y}} = {b}_{0} + {b}_{1}({x}_{1})^1+ {b}_{2}({x}_{1})^2+ {b}_{3}({x}_{1})^3$
- The graphs change so much as you go higher and higher in degree. It can result in a better fit if you chose the correct value.
- In all cases, the relationship between the variable and parameter is always linear.
- You can also have multi dimensional polynomial linear regression:
    - Numpy's polyfit function cannot perform this, so instead use the preprocessing library in scikit learn.
    - ${\hat{y}} = {b}_{0} + {b}_{1}{x}_{1}+ {b}_{2}{x}_{2}+ {b}_{3}{x}_{1}{x}_{2} + {b}_{4}({x}_{1})^2 + {b}_{5}({x}_{2})^2 +...$

How to create polynomial regression:

1. calculate polynomial of zth order:
    - f.np.polyfit(x, y, z)
    - p = np.poly1d(f)
2. print out the model:
    - print(p)

How to create multidimensional polynomial linear regression:

1. import preprocessing library
    - from sklearn.preprocessing import PolynomialFeatures
2. Create a polynomialfeature object
    - pr = polynomialfeatures(degree = 2, include_bias = False)
3. Transform the features into a polynomialfeature with the fit_transform method
    - x_polly = pr.fit_transform(x[['col1', 'col2']])
4. as the dimensions get larger, we may want to normalize multiple features in scikit-learn:
    - import library
        - from sklearn.preprocessing import StandardScaler
    - train the object
        - SCALE = StandardScaler()
    - fit the scale object
        - SCALE.fit(x_data[['col1', 'col2']])
    - transform the data into a new dataframe
        - x_scale = SCALE.transform(x_data[['col1', 'col2']])

Pipelines:

- There are many steps to getting a prediction
- Pipelines can help simplify the code by performing a series of transformations before the last step, which carries out the prediction.

- This method:
  - normalizes the data, performs a polynomial transformation, and outputs a prediction.

How to set up pipelines:

1. Import all needed modules
   - from sklearn.preprocessing import PolynomialFeatures
   - from sklearn.linear_model import LinearRegression
   - from sklearn.preprocessing import StandardScaler
   - from sklearn.pipeline import Pipeline
2. Create a list of tuples that contain the (estimatorModelName, ModelConstructor)
   - input = [('scale', StandardScaler()), ('polynomial', PolynomialFeatures(degree = 2), ..., ('mode', LinearRegression())]
3. Input the list into the pipeline constructor to get a pipeline object
   - pipe = Pipeline(input)
4. train the pipeline object
   - pipe.fit(df[['col1', ... , 'coln']], y)
5. Produce a prediction
   - yhat = pipe.prediction(x[['col1', ..., 'col4']])

Measures for In-Sample Evaluation

Now that we've seen how we can evaluate a model via visualization, we want to numerically evaluate models.

Measures used for in-sample evaluation:

- used to numerically determined how good the model fits on the dataset
- Two important measures to determine the fit of a model:
  - mean squared error (MSE)
  - R-Squared (R^2)

MSE:

- find the difference between actual value y and predicted value y, then square it.
- Then take the mean or average of all the errors by adding them all together and dividing by the number of samples.
- MSE tells you how close a regression line is to a set of points.

Find MSE in Python:

1. import library
   - from sklearn.metrics import mean_squared_error
2. Use the mean squared function
   - mean_squared_error(df[yvariable], y_predict_simple_fit)

R-Squared:

- AKA the coefficient of determination
- is a measure to determine how close the data is to the fitted regression line
- Interpret R-Square (x100) as the percentage of the variation in response variable y that is explained by the variation in explanatory variables x.
- R^2: the percentage of variation of the target variable (Y) that is explained by the linear model.
- R^2 = 1- ((MSE of regression line)/(MSE of average of data))
- Usually takes on values between (0, 1)
- If the line is a good fit, the MSE of the regression line will be small and the MSE of the average of the data will be somwhat larger
  - meaning that R^2 = 1 - (a number very close to zero)
- If R^2 is close to zero, the line did not perform well.
- If R^2 is negative, it can be due to overfitting.

Find R-Squared in Python:

1. plug in x and y
   - x = df[[col]]
   - y = df[col2]
2. use lm.fit
   - lm.fit(x,y)
3. use lm.score to get the value of r-squared
   - lm.score(x,y)
   - say this number comes out to be .46591188. Then you can say that approximately 46% of the variation of price is explained by the simple linear model. The higher the score, the better the model fit.

Prediction and Decision Making

Determining a good model fit:

- look at a combination of:
  - do the predicted values make sense?
  - visualization
  - numerical measures for evaluation
  - comparing models

Generate a sequence of values in a specified range to get some predicted values

- import numpy as np
- new_input = np.arange(1, 101, 1).reshape(-1, 1)
  - starts at 1 and increments by 1 until hitting 100
- yhat = lm.predict(new_input)
- Does the output make sense?
- Visualize the data to understand it better.
- test the numerical measures.

Comparing Multiple Linear Regression and Simple Linear Regression:

- does a lower Mean Square Error imply better fit?
  - not necessarily
  - MSE for MLR will be smaller than MSE for SLR since the errors of the data will decrease when more variables are included in the model.
  - polynomial regression will also have a smaller MSE than regular linear regression.

Week 5: Model Evaluation

Model Evaluation and Refinement

Model Evaluation

- In-sample evaluation tells us how well the model will fit the data used to train it.
- Problem: in-sample evaluation does not tell us how well the trained model can be used to predict new data.
- Solution:
  - in-sample data or training data
  - out of sample evaluation or test set
- Separating data into training and testing sets is an important part of model evaluation.
- Split data into:
  - Training set (estimate 70%)
  - Testing set (estimate 30%)
- Build and train the model with the training set
- Use testing set to asses the performance of a predictive model
- When we have completed testing the model, we should use all the data to train the model to get the best performance.

Splitting Data Sets in Python:

- SciKit-learn function train_test_split()
  - splits data into random train and test subsets
- Syntax:
  - from sklearn.model_selection import train_test_split
  - x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size = .3, random_state = 0)
    - x_data: features or independent variables
    - y_data: dataset target (such as price)
    - x_test, y_test: parts of available data as testing set
    - test_size: percentage of data used for testing
    - random_state: number generator used for random sampling

Generalization performance:

- Generalization error is a measure of how well the data does at predicting previously unseen data
- The error we obtain using the testing data is an approximation of this error.

Cross Validation

Cross Validation:

- Most common out of sample evaluation metrics
- More effective use of data (each observation is used for both training and testing)
- The data is divided into 'folds', with a portion of the folds used for training and the others used for testing.
- The process is repeated until each of the folds has had the opportunity to be both a test and train fold.
- Then the average results are used as an estimate of the out-of-sample error.
- In Python use cross_val_score():
  - from sklearn.model_selection import cross_val_score
  - scores = cross_val_score(lr, x_data, y_data, cv=3)
    - lr: linear regression model chosen here, but other model types can be used
    - x_data: the predictive variable data
    - y_data: the target variable data
    - cv: how many folds / splits we have in the data.
  - This function returns an array of scores, one for each partition that was chosen as the testing set.
  - We can average the result together to estimate out of sample r-squared using the mean function of NumPy
    - np.mean(scores)
- If we want to know the actual predicted values supplied by the model, use cross_val_predict()
  - it returns the prediction that was obtained for each element when it was in the test set.
  - Has a similar interface to cross_val_score()
  - from sklearn.model_selection import cross_val_predict
  - yhat = cross_val_predict(lr2e, x_data, y_data, cv = 3)

Overfitting and Underfitting

This section is specifically in regards to polynomial regression and how to pick the best polynomial order for your model.

Underfitting:

- The model is too simplistic to fit the data

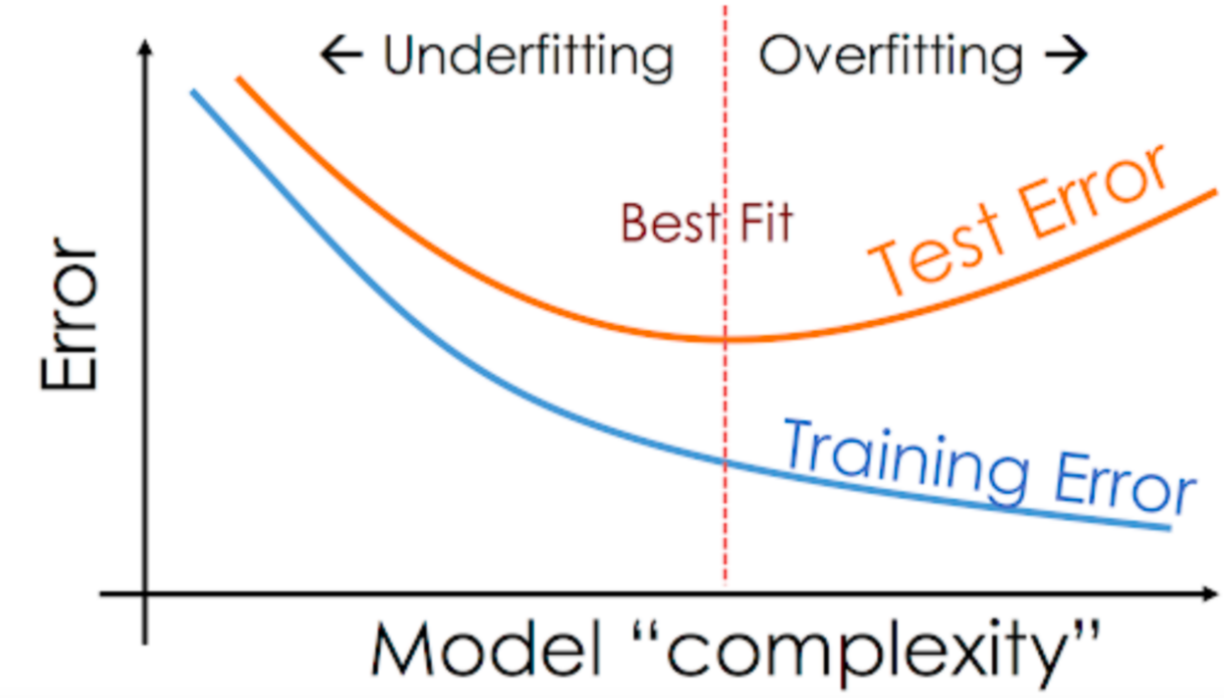
Overfitting:

- The model does well at tracking training points but performs poorly at estimating the function. This is especially apparent where there is little training data, where the estimated function will oscillate and not track the function probperly.

- The model is too flexible and fits the noise rather than the function.

Model Selection:

- On a graph comparing the Error MSE vs the order of the polynomial, we note:
- Training error decreases as the order of the polynomial goes up.
- Test error is a better means fo estimating the error of a polynomial. This error decreases until the best order of the polynomial is determined, then increases.
- Select the error that best minimizes the test error.
- Anything on the "left" of this graph is underfitting, while everything on the "right" is overfitting.
- Even the best choice for polynomial order will still have some error due to noise and other issues that is unavoidable.



Calculate different R-squared values in Python:

- Rsqu_test[]
- order = [1, 2, 3, 4]
- for n in order:
  - pr = POlynomialFeatures(degree = n)
  - x_train_pr=pr.fit_transform(x_train[['col']])
  - x_test_pr = pr.fit_transform(x_test[['col']])
  - lr.fit(x_train_pr, y_train
  - rsqu_test.append(lr.score(x_test_pr, y_test)

Ridge Regression

Ridge Regression:

- a regression that is employed in a multiple regression model when Multicollinearity occurs.

- Multicollinearity is when there is a strong relationship among the independent variables.
- Ridge regression is very common with polynomial regression.
- Ridge regression helps prevent overfitting, which is a big problem when you have multiple independent variables.
- Ridge regression helps to control the magnitude of the polynomial coefficients by introducing the parameter alpha, which is a parameter we select prior to fitting or training the model.
- if alpha is too large, the coefficients will approach zero and underfit the data.
- if alpha is zero, overfitting becomes a problem.
- Select the correct alpha using cross validation:
  - from sklearn.linear_model import Ridge
  - RidgeModel = Ridge(alpha = .1)
  - RidgeModel.fit(X,y)
  - yhat = RidgeModel.predict(X)
- To make a prediction, use the predict method. In order to determine alpha, use some data for training and a second set called validation data (which is similar to test data, but used to select parameters).
- Start with a small value of alpha, train the model, make a prediction using the validation data, and calculate the R-squared and store the values.
- Repeat this process for larger values of Alpha.
- Select the value of Alpha that maximizes R-squared

Grid Search

Grid Search:

- Allows us to scan through multiple free parameters with few lines of code.
- Hyperparameters:
  - alpha is a hyperparameter
  - Scikit-learn has a means of automatically iterating over these hyperparameters using cross-validation called Grid Search.
- Grid search takes the model or objects you want to train and different values of hyperparamters. It then calculates the MSE or R-squared for various hyperparameter values, allowing you to choose the best values.
- Select the hyperparameter that results in the smallest error out of all the grid search validation result errors.
- syntax:
  - from sklearn.linear_model import Ridge
  - from sklearn.model_selection import GridSearchCV
  - parameters = [{'alpha': [.001, .1, 1, 10, 100, 1000, 10000, 100000, 1000000], 'normalize': [True, False]}]
  - RR = Ridge()
  - Grid1 = GridSearchCV(RR, parameters, cv = 4)
  - Grid1.fit(x_data[['col1', 'col2', 'col3'...]], y_data)
  - Grid1.best*estimator*
  - scores = Grid1.cv*results*

    - scores['mean_test_score']
- Note: you can use the normalize parameter to help you decide if normalization is a good choice for your model.