Data Analysis with Python

|  |
| --- |
| Python Packages for Data Science  *A Python library is a collection of functions and methods that allow you to perform lots of actions without writing any code. The libraries usually contain built in modules providing different functionalities which you can use directly. There are extensive libraries offering a broad range of facilities. Libraries are broadly divided in three groups.*  Scientifics computing libraries:   * + **Pandas:** offers *data structure and tools* for effective data manipulation and analysis. Offers data structure and tools for effective data manipulation and analysis. It provides fast access to structured data. The primary instrument of Pandas is a two-dimensional table consisting of columns and rows labels which are called a DataFrame. It is designed to provide an easy indexing function   + **Numpy:** Uses arrays as their inputs and outputs. It can be extended to objects for matrices, and with a little change of coding, developers perform fast array processing Arrays & Matrices   + **SciPy:** includes functions for some advanced math problems as listed here, as well as data visualization. Integrals, solving differential equations and optimisation   Visualization libraries: *These libraries enable you to create graphs, charts and maps*   * + **Matplotlib:** The Matplotlib package is the most well-known library for data visualization. The graphs are also highly customizable.   + **Seaborn:** Another high-level visualization library is Seaborn. it is based on Matplotlib. It's very easy to generate various plots such as heat maps, time series and violin plots.   Machine learning algorithms: we're able to develop a model using our data set and obtain predictions. Here we introduce two packages   * + **Scikit-learn:** the Scikit-learn library contains tools statistical modelling, including regression, classification, clustering, and so on. This library is built on NumPy, SciPy and Matplotib.   + **Statsmodels:** is also a Python module that allows users to explore data, estimate statistical models, and perform statistical tests. |
| Import Data to Python  # Improtant working directory and function  Import os  os.getcwd() # For further information check this link - <https://note.nkmk.me/en/python-os-getcwd-chdir/>  # Import pandas library  import pandas as pd  # Read the online file by the URL provided, and assign it to variable "df"  other\_path = <https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DA0101EN/auto.csv>  df = pd.read\_csv(other\_path)  # show the first 5 rows using dataframe.head() method  print("The first 5 rows of the dataframe")  df.head(5)  # show the last 10 rows using dataframe.tail() method  print("The last 10 rows of the dataframe\n")  df.tail(10)  # create headers list  headers = ["symboling","normalized-losses","make","fuel-type","aspiration", "num-of-doors","body-style",  "drive-wheels","engine-location","wheel-base", "length","width","height","curb-weight","engine-type",  "num-of-cylinders", "engine-size","fuel-system","bore","stroke","compression-ratio","horsepower",  "peak-rpm","city-mpg","highway-mpg","price"]  print("headers\n", headers)  # We replace headers and recheck our data frame  df.columns = headers  df.head(10)  # We can drop missing values along the column "price" as follows  df.dropna(subset=["price"], axis=0)  # Write your code below  print(df.columns)  df.price – for particular column  df.price[0:10] – for particular column and rows 0 to 9  df[['price']] – as table select  df[['price']].head()  *Correspondingly, Pandas enables us to save the dataset to csv by using the dataframe.to\_csv() method, you can add the file path and name along with quotation marks in the brackets.*  *For example, if you would save the dataframe df as automobile.csv to your local machine, you may use the syntax below:*  df.to\_csv("automobile.csv", index=False) # saves to current directory   |  |  |  | | --- | --- | --- | | 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() | | hdf | pd.read\_hdf() | df.to\_hdf() | | sql | pd.read\_sql() | df.to\_sql() | |
| Basic Insight of Dataset  Understand your data before you begin any ananlysis  After reading data into Pandas dataframe, it is time for us to explore the dataset. We should check following:   * Data Types * Data Distribution * Locate potential issues with the data   There are several ways to obtain essential insights of the data to help us better understand our dataset.  **Data Types:**  Data has a variety of types. The main types stored in Pandas dataframes are object, float, int, bool and datetime64. In order to better learn about each attribute, it is always good for us to know the data type of each column.  # check the data type of data frame "df" by .dtypes  print(df.dtypes)   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | symboling | int64 | engine-location | object | engine-size | int64 |   *As a result, as shown above, it is clear to see that the data type of "symboling" and "curb-weight" are int64, "normalized-losses" is object, and "wheel-base" is float64, etc.*  *These data types can be changed; we will learn how to accomplish this in a later module.*    **Describe:** *If we would like to get a statistical summary of each column, such as count, column mean value, column standard deviation, etc. We use the describe method: dataframe.describe()*  *This method will provide various summary statistics, excluding NaN (Not a Number) values.*  df.describe()  *You can add an argument include = "all" inside the bracket. Let's try it again.*  df.describe(include = "all")  *Now, it provides the statistical summary of all the columns, including object-typed attributes. We can now see how many unique values, which is the top value and the frequency of top value in the object-typed columns. Some values in show as "NaN", this is because those numbers are not available regarding a particular column type.*  *You can select the columns of a data frame by indicating the name of each column, for example, you can select the three columns as follows:*dataframe[[' column 1 ',column 2', 'column 3']]  # Where "column" is the name of the column, you can apply the method ".describe()" to get the statistics:  dataframe[[' column 1 ',column 2', 'column 3'] ].describe()  # Apply the method to ".describe()" to the columns 'length' and 'compression-ratio':  df[['length','compression-ratio']].describe()  **Info:** Another method you can use to check your dataset is: dataframe.info  # look at the info of "df"  df.info  Here we are able to see the information of our dataframe, with the top 30 rows and the bottom 30 rows.And, it also shows us the whole data frame has 205 rows and 26 columns in total  **Summary:**  df.dtypes to check datatypes  df.describe() – gives stats on numerical data  df.describe(include = "all") – stats on all data  df.info() – Shows top 30 andbottom 30 rows of datafram  df.dropna(subset=["price"], axis=0) |
| Data Wrangling (Data Cleansing or Pre-processing)  Data Wrangling is the process of converting data from the initial format to a format that may be better for analysis.  Identify Missing Values  In the car dataset, missing data comes with the question mark "?". We replace "?" with NaN (Not a Number), which is Python's default missing value marker, for reasons of computational speed and convenience. Here we use the function: .replace(A, B, inplace = True)  # Import function numpy  import numpy as np  # replace "?" to NaN  df.replace("?", np.nan, inplace = True)  df.head(5)  Evaluating for Missing Data:  The missing values are converted to Python's default. We use Python's built-in functions to identify these missing values. There are two methods to detect missing data: .isnull(), .notnull()  The output is a boolean value (TRUE/FALSE) indicating whether the value that is passed into the argument is in fact missing data.  missing\_data = df.isnull()  missing\_data.head(5)   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | symboling | normalized-losses | make | fuel-type | aspiration | | 0 | FALSE | TRUE | FALSE | FALSE | FALSE | | 1 | FALSE | TRUE | FALSE | FALSE | FALSE | | 2 | FALSE | TRUE | FALSE | FALSE | FALSE | | 3 | FALSE | FALSE | FALSE | FALSE | FALSE | | 4 | FALSE | FALSE | FALSE | FALSE | FALSE |   Here, "True" stands for missing value, while "False" stands for not missing value.  Count missing values in each column:  # Using a for loop in Python, we can quickly figure out the number of missing values in each column. As mentioned above, "True" represents a missing value, "False" means the value is present in the dataset. In the body of the for loop the method ".value\_counts()" counts the number of "True" values.  for column in missing\_data.columns.values.tolist():   |  |  | | --- | --- | | symboling  False 205  Name: symboling, dtype: int64 | normalized-losses  False 164, True 41  Name: normalized-losses, dtype: int64 | | make  False 205  Name: make, dtype: int64 | fuel-type  False 205  Name: fuel-type, dtype: int64 |   print(column)  print (missing\_data[column].value\_counts())  print("")  Based on the summary, each column has 205 rows, and seven columns containing missing data:   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | "normalized-losses": 41 | "stroke" : 4 | "num-of-doors": 2 | "bore": 4 | "horsepower": 2 missing | "peak-rpm": 2 | "price": 4 |   Drop column:  *Whole columns should be dropped only if most entries in the column are empty. In our dataset, none of the columns are empty enough to drop entirely.*  df.drop("ful-tpe", axis = 1, inplace=True) # note it’s for example, there is no column name “ful-tpe” in dataframe  Replace values:  *We have some freedom in choosing which method to replace data; however, some methods may seem more reasonable than others. We will apply each method to many different columns:*  **Replace by mean:**  *Replace missing data with mean for "normalized-losses", "stroke”, "bore", "horsepower", "peak-rpm"*  # Calculate the average of the column  avg\_norm\_loss = df["normalized-losses"].astype("float").mean(axis=0)  print("Average of normalized-losses:", avg\_norm\_loss)  # Replace "NaN" by mean value of "normalized-losses" column  df["normalized-losses"].replace(np.nan, avg\_norm\_loss, inplace=True)  # Calculate the mean value for 'bore' column  avg\_bore=df['bore'].astype('float').mean(axis=0)  print("Average of bore:", avg\_bore)  # Replace NaN by mean value of 'bore' column  df["bore"].replace(np.nan, avg\_bore, inplace=True)  **Replace by frequency:**  *Replace missing "num-of-doors" values with "four".*  *Reason: 84% sedans is four doors. Since four doors is most frequent, it is most likely to occur*  # To see which values are present in a particular column, we can use the ".value\_counts()" method:  df['num-of-doors'].value\_counts()  *We can see that four doors are the most common type. We can also use the ".idxmax()" method to calculate for us the most common type automatically:*  df['num-of-doors'].value\_counts().idxmax()  *The replacement procedure is very similar to what we have seen previously*  # Replace the missing 'num-of-doors' values by the most frequent  df["num-of-doors"].replace(np.nan, "four", inplace=True)  **Drop the whole row:**  *Finally, let's drop all rows that do not have price data i.e.where "price" is missing*  *Reason: price is what we want to predict. Any row without price data is not useful us, as it can’t be used for prediction.*  # Drop whole row with NaN in "price" column  df.dropna(subset=["price"], axis=0, inplace=True)  # Reset index, because we droped two rows  df.reset\_index(drop=True, inplace=True)  Correct data format:  *The last step in data cleaning is checking and making sure that all data is in the correct format (int, float, text or other).In Pandas, we use .dtype() to check the data type and .astype() to change the data type*  # Lets list the data types for each column: df.dtypes   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | normalized-losses | object | wheel-base | float64 | fuel-system | object | | make | object | length | float64 | bore | object | | fuel-type | object | width | float64 | stroke | object | | aspiration | object | height | float64 | compression-ratio | float64 | | body-style | object | engine-type | object | peak-rpm | object | | price | object | dtype: | object |  |  |   *As we can see, some columns are not of the correct data type. Numerical variables should have type 'float' or 'int', and variables with strings such as categories should have type 'object'.*  *For example, 'bore' and 'stroke' variables are numerical values that describe the engines, so we should expect them to be of the type 'float' or 'int’.*  # We have to convert data types into a proper format for each column using the "astype()" method.  df[["bore", "stroke","peak-rpm"] = df[["bore", "stroke","peak-rpm"]].astype("float")  df[["normalized-losses"]] = df[["normalized-losses"]].astype("int")  df[["price"]] = df[["price"]].astype("float") *# my practice update statement – dummydf[['price','bore']]=[0,1]* |
| Data Standardization:  *Data is usually collected from different agencies with different formats. Data Standardization is also a term for a particular type of data normalization, where we subtract the mean and divide by the standard deviation.*  What is Standardization?  *Standardization is the process of transforming data into a common format which allows the researcher to make the meaningful comparison.*  ***Data Transformation:***  *In our dataset, the fuel consumption columns "city-mpg" and "highway-mpg" are represented by mpg (miles per gallon) unit. Assume we are developing an application in a country that accept the fuel consumption with L/100km standard.*  *We can do many mathematical operations directly in Pandas.*  *Example: Transform mpg to L/100km:*   * *We will need to apply data transformation to transform mpg into L/100km* * *The formula for unit conversion is: L/100km = 235 / mpg*   # Convert mpg to L/100km by mathematical operation (235 divided by mpg) – New Column  df['city-L/100km'] = 235/df["city-mpg"]  # Transform mpg to L/100km by mathematical operation (235 divided by mpg) – Same Column  df["highway-mpg"] = 235/df["highway-mpg"]  # Rename column name from "highway-mpg" to "highway-L/100km"  df.rename(columns={'"highway-mpg"':'highway-L/100km'}, inplace=True)  ***Data Normalization:***  Why normalization?  *Normalization is the process of transforming values of several variables into a similar range. Typical normalizations include scaling the variable so the variable average is 0, scaling the variable so the variance is 1, or scaling variable so the variable values range from 0 to 1*  *Example: To demonstrate, let's say we want to scale the columns "length", "width" and "height"*   * *Target: Would like to Normalize these variables so their value ranges from 0 to 1.* * *There three ways/approch to do it. i.e. by division by maximum value, or by Zscore*   # Approach 1 : Replace original value by (original value)/(maximum value)  df['length'] = df['length']/df['length'].max()  # Approach 2 : Replace original value by (original value)/(maximum value)  df['width'] = df['width']/df['width'].max()  # Approach 3 : Z Score - replace original value by (original value-mean)/(std.)  (df["length"]-df["length"].mean())/df["length"].std()  *Here we can see, we've normalized "length", "width" and "height" in the range of [0,1].*  ***Binning:***  Why binning?  *Binning is a process of transforming continuous numerical variables into discrete categorical 'bins', for grouped analysis.*  *Example:*  *In our dataset, "horsepower" is a real valued variable ranging from 48 to 288, it has 57 unique values. What if we only care about the price difference between cars with high, medium, and little horsepower (3 types)? Can we rearrange them into three ‘bins' to simplify analysis?*   * *Convert data to correct format incase not in right format*   df["horsepower"]=df["horsepower"].astype(int, copy=True)   * *We will use numpy's linspace() function to creat bin array.,and then Pandas method 'cut' to segment the 'horsepower' column into 3 bins*   # Let’s creat 3 bins of equal size bandwidth using linspace(start\_value, end\_value, numbers\_generated)   * Since we want to include the minimum, maximum value, set start\_value = min() and end\_value = max() * Since we need 3 bins of equal length, there should be 4 dividers, so numbers\_generated=4   bins = np.linspace(min(df["horsepower"]), max(df["horsepower"]), 4)  bins – array([ 48.0, 119.33333333, 190.66666667, 262.0])  # We build a bin array, with a minimum value to a maximum value, with bandwidth calculated above. The bins will be values used to determine when one bin ends and another begins. We are using the one created in previouse step  bins = np.linspace(min(df["horsepower"]), max(df["horsepower"]), 4)  # We set group names which we will assign to our “bins” array ranges:  group\_names = ['Low', 'Medium', 'High']  # We apply the function "cut" the determine each value of "df['horsepower']" belongs to which bin:  df['HS-binned'] = pd.cut(df['horsepower'], bins, labels=group\_names, include\_lowest=True )  df[['horsepower','HS-binned']].head()  # Lets see the number of vehicles in each bin  df["HS-binned"].value\_counts()  *# Lets plot the distribution of each bin using bargraph:*  # Import matplotlib pacakge and pyplot module  %matplotlib inline  import matplotlib as plt  from matplotlib import pyplot  # Set X and Y axis values for bargraph  pyplot.bar(group\_names, df["HS-binned"].value\_counts())  # set x/y labels and plot title  plt.pyplot.xlabel("horsepower")  plt.pyplot.ylabel("count")  plt.pyplot.title("horsepower bins")  ***Bins visualization:***  *Normally, a histogram is used to visualize the distribution of bins we created above.*  *# Lets plot the distribution using* histograms  # Note - Import matplotlib pacakge and pyplot module if not available  # draw historgram of attribute "horsepower" with bins = 3  plt.pyplot.hist(df["horsepower"], bins = 3)  # set x/y labels and plot title  plt.pyplot.xlabel("horsepower")  plt.pyplot.ylabel("count")  plt.pyplot.title("horsepower bins")  *The plot on right shows the binning result for attribute "horsepower"*  ***Indicator or dummy variable:***  What is an indicator variable?  *An indicator variable (or dummy variable) is a numerical variable used to label categories. They are called 'dummies' because the numbers themselves don't have inherent meaning.*  Why we use indicator variables?  *So we can use categorical variables for regression analysis in the later modules.*  Example:  *We see the column "fuel-type" has two unique values, "gas" or "diesel". Regression doesn't understand words, only numbers. To use this attribute in regression analysis, we convert "fuel-type" into indicator variables.*  #Use the panda's method 'get\_dummies' to assign numerical values to different categories of fuel type.  df.columns *# Check columns*  dummy\_v\_1 = pd.get\_dummies(df["fuel-type"]) *# Indicator variable created*  dummy\_v\_1.head() *# Checking Indicator table*  # Change column names for clarity  dummy\_v\_1.rename(columns={'fuel-type-diesel':'gas', 'fuel-type-diesel':'diesel'}, inplace=True)  *We now have the value 0 to represent "gas" and 1 to represent "diesel" in the column "fuel-type". We will now insert this column back into our original dataset.*  # Merge data frame "df" and "dummy\_v\_1"  df = pd.concat([df, dummy\_v\_1], axis=1) *# Use of concat() function for dataframe*  df.drop("fuel-type", axis = 1, inplace=True) *# Drop original column "fuel-type" from "df"*  ***For a practice, create indicator variable to the column of "aspiration": "std" to 0, while "turbo" to 1.***  # Get indicator variables of aspiration  dummy\_v\_2 = pd.get\_dummies(df["aspiration"]) *# Crate and assign indicator to new dataframe*  dummy\_v\_2.rename(columns={'std':'aspiration-std', 'turbo': 'aspiration-turbo'}, inplace=True) *# change column names*  dummy\_v\_2.head() *# Checking Indicator table*  df = pd.concat([df, dummy\_v\_2], axis=1) *# Merge the new dataframe to the original datafram*  df.drop('aspiration', axis = 1, inplace=True) # drop original column "aspiration" from "df"  *Save the cleaned data new csv using pandas function dataframe to\_csv()*  df.to\_csv('clean\_df.csv')  Link for practice notebook: <https://labs.cognitiveclass.ai/tools/jupyterlab/lab/tree/labs/DA0101EN/data-wrangling.ipynb> |
| Exploratory Data Analysis  *Priliminary step in data analysis:*   * *Summarize main charactristics of data* * *Gain better understanding of data set* * *Uncover relationships between variables* * *Extract importnt variables*   *In this section, we will do exploratory data analysis i.e. we will explore few methods to see if certain characteristics or features of car can be used to predict car price. The purpose of the analysis is to address the following question:*  *“What are the main characteristics which have the most impact on the car price?”*  *Learning Objectives:*   * *Descriptive Statistics* * *Basics of Grouping* * *ANOVA – Analysis of Variance* * *Correlation and Correlation Statistics*   ***Import Data from Module:***  # Import clean data from earlier module  path='https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DA0101EN/automobileEDA.csv'  path = '/resources/labs/DA0101EN/clean\_df.csv'  df = pd.read\_csv(path)  import pandas as pd *# Import libraries*  import numpy as np  ***Analyzing Individual Feature Patterns using Visualization:***  # To install seaborn we use the pip, which is the python package manager.  %%capture  ! pip install seaborn *# install seaborn*  # Import visualization packages  import matplotlib.pyplot as plt *# Import "Matplotlib" and "Seaborn"*  import seaborn as sns  %matplotlib inline *# Don't forget about "%matplotlib inline" required to plot in a Jupyter notebook*  How to choose the right visualization method?  *When visualizing individual variables, it is important to first understand what type of variable you are dealing, using* print (df.dtypes)*. This will help us find the right visualization method for that variable.*  *Example:*  *We can calculate the correlation between variables of type "int64" or "float64" using the method "corr”:*  df.corr() *# Correlation between each feature(column) in data*  df[['bore','stroke' ,'compression-ratio','horsepower']].corr() *# Correlation between bore, stroke, compression-ratio & HP*   |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | **bore** | **stroke** | **compression-ratio** | **horsepower** | | **bore** | 1.000000 | -0.055390 | 0.001263 | 0.566936 | | **stroke** | -0.055390 | 1.000000 | 0.187923 | 0.098462 | | **compression-ratio** | 0.001263 | 0.187923 | 1.000000 | -0.214514 |   *Correlation is a measure of the extent of interdependence between variables*  *Note: The diagonal elements are always one in output table*  Continuous numerical variables:  *Continuous numerical variables are variables that may contain any value within some range. Continuous numerical variables can have the type "int64" or "float64". A great way to visualize these variables is by using scatterplots with fitted lines.*  *In order to start understanding the (linear) relationship between an individual variable and the price. We can do this by using "regplot", which plots the scatterplot plus the fitted regression line for the data.*  *Let's see several examples of different linear relationships:*  Positive linear relationship  *Let's find the scatterplot of "engine-size" and "price"*  # Engine size as potential predictor variable of price  sns.regplot(x="engine-size", y="price", data=df)  *As the engine-size goes up, the price goes up: this indicates a positive direct correlation between these two variables. Engine size seems like a pretty good predictor of price since the regression line is almost a perfect diagonal line.*  df[["engine-size", "price"]].corr()   |  |  |  | | --- | --- | --- | |  | engine-size | price | | engine-size | 1.000000 | 0.872335 | | price | 0.872335 | 1.000000 |   *We can examine the correlation between 'engine-size' and 'price' and see it's approximately 0.87*  Negative linear relationship  # Find out if Highway-mpg is a potential predictor variable of price  sns.regplot(x="highway-mpg", y="price", data=df)  *As the highway-mpg goes up, the price goes down: this indicates an inverse/negative relationship between these two variables. Highway mpg could potentially be a predictor of price.*  df[['highway-mpg', 'price']].corr()   |  |  |  | | --- | --- | --- | |  | highway-mpg | price | | highway-mpg | 1.000000 | -0.704692 | | price | -0.704692 | 1.000000 |   *We can examine the correlation between 'highway-mpg' and 'price' and see it's approximately -0.704*  Weak Linear Relationship  # Let's see if "Peak-rpm" as a predictor variable of "price".  sns.regplot(x='peak-rpm', y='price', data=df)  *Peak rpm does not seem like a good predictor of the price at all since the regression line is close to horizontal. Also, the data points are very scattered and far from the fitted line, showing lots of variability. Therefore, it's it is not a reliable variable.*  df[['peak-rpm','price']].corr()   |  |  |  | | --- | --- | --- | |  | peak-rpm | price | | peak-rpm | 1.000000 | -0.101616 | | price | -0.101616 | 1.000000 |   We can examine the correlation between 'peak-rpm' and 'price' and see it's approximately -0.101616  # Find the correlation between x="stroke", y="price".  df[["stroke","price"]].corr()  *Given the correlation results between "price" and "stroke" do you expect a linear relationship?*  sns.regplot(x="stroke", y="price", data=df)  *Conclusion:* |
| Categorical variables  *These are variables that describe a 'characteristic' of a data unit, and are selected from a small group of categories. The categorical variables can have the type "object" or "int64". A good way to visualize categorical variables is by using boxplots.*    # Let's look at the relationship between "body-style" and "price".  sns.boxplot(x="body-style", y="price", data=df)  *We see that the distributions of price between the different body-style categories have a significant overlap, and so body-style would not be a good predictor of price.*    # Let's examine engine "engine-location" and "price"  sns.boxplot(x="engine-location", y="price", data=df)  *Here we see that the distribution of price between these two engine-location categories, front and rear, are distinct enough to take engine-location as a potential good predictor of price.*  # Let's examine "drive-wheels" and "price".  sns.boxplot(x="drive-wheels", y="price", data=df)  *Here we see that the distribution of price between the different drive-wheels categories differs; as such drive-wheels could potentially be a predictor of price.*  ***Descriptive Statistical Analysis:***  *Let's first take a look at the variables by utilizing a description method. The describe function automatically computes basic statistics for all continuous variables. Any NaN values are automatically skipped in these statistics.*  # We can apply the method "describe" as follows:  df.describe()  This will show:   * 1. the count of that variable   2. the mean   3. the standard deviation (std)   4. the minimum value   5. the IQR (Interquartile Range: 25%, 50% and 75%)   6. the maximum value   *The default setting of "describe" skips variables of type object.* *We can include the variables of type 'object' as follows:*  df.describe(include=['object'])  This will show:   * 1. the count of that variable   2. the unique count of that variable   3. top – mode value of variable i.e. based on occurrence   4. freq – occurrence of mode value i.e. for the variable value captured as top occurrence   Value Counts  *Value-counts is a good way of understanding how many units of each characteristic/variable we have. Let’s apply the method on the column 'drive-wheels'. Note: The method "value\_counts" only works on Pandas series, not Pandas dataframes. As a result, we only include one bracket "df['drive-wheels']" not two brackets "df[['drive-wheels']]".*  # Use value\_counts()  df['drive-wheels'].value\_counts()  fwd 118  rwd 75  4wd 8  Name: drive-wheels, dtype: int64   |  |  | | --- | --- | |  | drive-wheels | | fwd | 118 | | rwd | 75 | | 4wd | 8 |   # We can convert the series to a Dataframe as follows:  df['drive-wheels'].value\_counts().to\_frame()  # Let's repeat the above steps but save the results to the dataframe "drive\_wheels\_counts" and rename the column 'drive-wheels' to 'value\_counts'.  drive\_wheels\_counts = df['drive-wheels'].value\_counts().to\_frame()   |  |  | | --- | --- | |  | value\_counts | | fwd | 118 | | rwd | 75 | | 4wd | 8 |   drive\_wheels\_counts.rename(columns={'drive-wheels': 'value\_counts'}, inplace=True)  drive\_wheels\_counts   |  |  | | --- | --- | | drive-wheels | value\_counts | | fwd | 118 | | rwd | 75 | | 4wd | 8 |   # Now let's rename the index to 'drive-wheels':  drive\_wheels\_counts.index.name = 'drive-wheels'  drive\_wheels\_counts  *We can repeat the above process for the variable 'engine-location'.*   |  |  | | --- | --- | | engine-location | value\_counts | | front | 198 | | rear | 3 |   # Engine-location as variable  engine\_loc\_counts = df['engine-location'].value\_counts().to\_frame()  engine\_loc\_counts.rename(columns={'engine-location': 'value\_counts'}, inplace=True)  engine\_loc\_counts.index.name = 'engine-location'  *Examining the value counts of the engine location would not be a good predictor variable for the price. This is because we only have three cars with a rear engine and 198 with an engine in the front, this result is skewed. Thus, we are not able to draw any conclusions about the engine location.* |
| ***Basics of Grouping***  *The "groupby" method groups data by different categories. The data is grouped based on one or several variables and analysis is performed on the individual groups.*  *For example, let's group by the variable "drive-wheels". We see that there are 3 different categories of drive wheels.*  df['drive-wheels'].unique()  df  # array(['rwd', 'fwd', '4wd'], dtype=object)  *If we want to know, on average, which type of drive wheel is most valuable, we can group "drive-wheels" and then average them.*  # Select the columns 'drive-wheels', 'body-style' and 'price', then assign it to the variable  df\_group\_one = df[['drive-wheels','body-style','price']]  # Group results to calculate the average price for each of the different categories of data  df\_group\_one = df\_group\_one.groupby(['drive-wheels'],as\_index=False).mean()  df\_group\_one   |  |  |  | | --- | --- | --- | |  | drive-wheels | price | | 0 | 4wd | 10241.000000 | | 1 | fwd | 9244.779661 | | 2 | rwd | 19757.613333 |   *From our data, it seems rear-wheel drive vehicles are, on average, the most expensive, while 4-wheel and front-wheel are approximately the same in price.*  # Group multiple variables  *For example, let's group by both 'drive-wheels' and 'body-style'. This groups the dataframe by the unique combinations 'drive-wheels' and 'body-style'. We can store the results in the variable 'grouped\_test1'.*  df\_gptest = df[['drive-wheels','body-style','price']]  grouped\_test1 = df\_gptest.groupby(['drive-wheels','body-style'],as\_index=False).mean()  grouped\_test1     |  |  |  |  | | --- | --- | --- | --- | |  | drive-wheels | body-style | price | | 0 | 4wd | hatchback | 7603.000000 | | 1 | 4wd | sedan | 12647.333333 | | 2 | 4wd | wagon | 9095.750000 | | 3 | fwd | convertible | 11595.000000 | | 4 | fwd | hardtop | 8249.000000 | | .. | … | … | …. |   201 rows × 3 columns  *This grouped data is much easier to visualize when it is made into a pivot table. A pivot table is like an Excel spreadsheet, with one variable along the column and another along the row. We can convert the dataframe to a pivot table using the method "pivot " to create a pivot table from the groups.*  *In this case, we will leave the drive-wheel variable as the rows, and pivot body-style to become the columns of the table:*  grouped\_pivot = grouped\_test1.pivot(index='drive-wheels', columns='body-style')  grouped\_pivot   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | price | body-style |  |  |  |  | | drive-wheels | convertible | hardtop | hatchback | sedan | wagon | | 4wd | NaN | NaN | 7603.000000 | 12647.333333 | 9095.750000 | | fwd | 11595.0 | 8249.000000 | 8396.387755 | 9811.800000 | 9997.333333 | | rwd | 23949.6 | 24202.714286 | 14337.777778 | 21711.833333 | 16994.222222 |   *Often, we won't have data for some of the pivot cells. We can fill these missing cells with the value 0, but any other value could potentially be used as well. It should be mentioned that missing data is quite a complex subject and is an entire course on its own.*  grouped\_pivot = grouped\_pivot.fillna(0) #fill missing values with 0  grouped\_pivot   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | price | body-style |  |  |  |  | | drive-wheels | convertible | hardtop | hatchback | sedan | wagon | | 4wd | 0.0 | 0.000000 | 7603.000000 | 12647.333333 | 9095.750000 | | fwd | 11595.0 | 8249.000000 | 8396.387755 | 9811.800000 | 9997.333333 | | rwd | 23949.6 | 24202.714286 | 14337.777778 | 21711.833333 | 16994.222222 |   # Use the "groupby" function to find the average "price" of each car based on "body-style”?  grouped\_test\_bodystyle = df[['body-style', 'price']].groupby(['body-style'],as\_index= False).mean()  grouped\_test\_bodystyle  *Let’s plot Heatmap for variables: Drive Wheels and Body Style vs Price:*  # If you did not import "pyplot" let's do it again.  import matplotlib.pyplot as plt  %matplotlib inline  plt.pcolor(grouped\_pivot, cmap='RdBu') *# Use the grouped results*  plt.colorbar()  plt.show()  *The heatmap plots the target variable (price) proportional to colour with respect to the variables 'drive-wheel' and 'body-style' in the vertical and horizontal axis respectively. This allows us to visualize how the price is related to 'drive-wheel' and 'body-style'.*  # The default labels convey no useful information to us. Let's change that:  fig, ax = plt.subplots()  im = ax.pcolor(grouped\_pivot, cmap='RdBu')  ​row\_labels = grouped\_pivot.columns.levels[1] *# Label names*  col\_labels = grouped\_pivot.index  ​ax.set\_xticks(np.arange(grouped\_pivot.shape[1]) + 0.5, minor=False) *# Move ticks and labels to the center*  ax.set\_yticks(np.arange(grouped\_pivot.shape[0]) + 0.5, minor=False)  ​ax.set\_xticklabels(row\_labels, minor=False) # Insert labels  ax.set\_yticklabels(col\_labels, minor=False)  ​plt.xticks(rotation=90) *# Rotate label if too long*  fig.colorbar(im)  plt.show()  *Visualization is very important in data science, and Python visualization packages provide great freedom. We will go more in-depth in a separate Python Visualizations course.*  ***Correlation and Causation***  *To find out the main characteristics which have the most impact on the car price and get a better measure of the important characteristics, we look at the correlation of these variables with the car price, in other words: how is the car price dependent on this variable?*  *Correlation: a measure of the extent of interdependence between variables.*  *Causation: the relationship between cause and effect between two variables.*  *It is important to know the difference between these two and that correlation does not imply causation. Determining correlation is much simpler the determining causation as causation may require independent experimentation.*  Pearson Correlation  *The Pearson Correlation measures the linear dependence between two variables X and Y. The resulting coefficient is a value between -1 and 1 inclusive, where:*   |  | | --- | | 1 : Total positive linear correlation. | | 0 : No linear correlation, the two variables most likely do not affect each other. | | -1 : Total negative linear correlation. |   *Pearson Correlation is the default method of the function "corr". Like before we can calculate the Pearson Correlation of the of the 'int64' or 'float64' variables.* df.corr()  P-value: *Shows the statistical significance of the correlation estimate*  *What is this P-value? The P-value is the probability value that the correlation between these two variables is statistically significant. Normally, we choose a significance level of 0.05, which means that we are 95% confident that the correlation between the variables is significant. By convention, when the p-value is*:   |  | | --- | | < 0.001 : we say there is strong evidence that the correlation is significant. | | < 0.05 : there is moderate evidence that the correlation is significant. | | < 0.1 : there is weak evidence that the correlation is significant. | | > 0.1 : there is no evidence that the correlation is significant. |   # We can obtain this information using “stats” module in the “scipy” library.  From scipy import stats   |  | | --- | | # Wheel-base vs Price: Let’s calculate the Pearson Correlation Coefficient and P-value of ‘wheel-base’ and ‘price’.  Pearson\_coef, p\_value = stats.pearsonr(df[‘wheel-base’], df[‘price’])  print(“The Pearson Correlation Coefficient is”, pearson\_coef, “ with a P-value of P =”, p\_value)  The Pearson Correlation Coefficient is 0.584641822265508 with a P-value of P = 8.076488270733218e-20  *Conclusion:* *Since the p-value is < 0.001, the correlation between wheel-base and price is statistically significant, although the linear relationship isn’t extremely strong (~0.585)* | | # Horsepower vs Price: Let’s calculate the Pearson Correlation Coefficient and P-value of ‘horsepower’ and ‘price’.  Pearson\_coef, p\_value = stats.pearsonr(df[‘horsepower’], df[‘price’])  print(“The Pearson Correlation Coefficient is”, pearson\_coef, “ with a P-value of P = “, p\_value)  The Pearson Correlation Coefficient is 0.809574567003656 with a P-value of P = 6.369057428259557e-48  *Conclusion: Since the p-value is < 0.001, the correlation between horsepower and price is statistically significant, and the linear relationship is quite strong (~0.809, close to 1)* | | # Length vs Price: Let’s calculate the Pearson Correlation Coefficient and P-value of ‘length’ and ‘price’.  Pearson\_coef, p\_value = stats.pearsonr(df[‘length’], df[‘price’])  print(“The Pearson Correlation Coefficient is”, pearson\_coef, “ with a P-value of P = “, p\_value)  The Pearson Correlation Coefficient is 0.690628380448364 with a P-value of P = 8.016477466158986e-30  *Conclusion: Since the p-value is < 0.001, the correlation between length and price is statistically significant, and the linear relationship is moderately strong (~0.691)* | | # Width vs Price: Let’s calculate the Pearson Correlation Coefficient and P-value of ‘width’ and ‘price’.  Pearson\_coef, p\_value = stats.pearsonr(df[‘width’], df[‘price’])  print (“The Pearson Correlation Coefficient is”, pearson\_coef, “ with a P-value of P =”, p\_value )  The Pearson Correlation Coefficient is 0.7512653440522674 with a P-value of P = 9.200335510481516e-38  *Conclusion: Since the p-value is < 0.001, the correlation between width and price is statistically significant, and the linear relationship is quite strong (~0.751).* | | # Curb-weight vs Price: Let's calculate the Pearson Correlation Coefficient and P-value of 'curb-weight' and 'price'.  pearson\_coef, p\_value = stats.pearsonr(df['curb-weight'], df['price'])  print ("The Pearson Correlation Coefficient is", pearson\_coef, " with a P-value of P = ", p\_value)  The Pearson Correlation Coefficient is 0.8344145257702846 with a P-value of P = 2.1895772388936914e-53  *Conclusion: Since the p-value is < 0.001, the correlation between curb-weight and price is statistically significant, and the linear relationship is quite strong (~0.834)* | | # Engine-size vs Price: Let's calculate the Pearson Correlation Coefficient and P-value of 'engine-size' and 'price'.  pearson\_coef, p\_value = stats.pearsonr(df['engine-size'], df['price'])  print ("The Pearson Correlation Coefficient is", pearson\_coef, " with a P-value of P =", p\_value)  The Pearson Correlation Coefficient is 0.8723351674455185 with a P-value of P = 9.265491622198389e-64  *Conclusion: Since the p-value is < 0.001, the correlation between engine-size and price is statistically significant, and the linear relationship is very strong (~0.872)* |  |  | | --- | | # Bore vs Price: Let's calculate the Pearson Correlation Coefficient and P-value of 'bore' and 'price'.  pearson\_coef, p\_value = stats.pearsonr(df['bore'], df['price'])  print ("The Pearson Correlation Coefficient is", pearson\_coef, " with a P-value of P = ", p\_value)  The Pearson Correlation Coefficient is 0.5431553832626601 with a P-value of P = 8.049189483935489e-17  *Conclusion: Since the p-value is < 0.001, the correlation between bore and price is statistically significant, but the linear relationship is only moderate (~0.521)* | | # City-mpg vs Price: We can relate the process for each 'City-mpg' and 'Highway-mpg':  pearson\_coef, p\_value = stats.pearsonr(df['city-mpg'], df['price'])  print ("The Pearson Correlation Coefficient is", pearson\_coef, " with a P-value of P = ", p\_value)  The Pearson Correlation Coefficient is -0.6865710067844677 with a P-value of P = 2.321132065567674e-29  *Conclusion: Since the p-value is < 0.001, the correlation between city-mpg and price is statistically significant, and the coefficient of ~ -0.687 shows that the relationship is negative and moderately strong*. | | Highway-mpg vs Price:  pearson\_coef, p\_value = stats.pearsonr(df['highway-mpg'], df['price'])  print ("The Pearson Correlation Coefficient is", pearson\_coef, " with a P-value of P = ", p\_value)  The Pearson Correlation Coefficient is -0.7046922650589529 with a P-value of P = 1.7495471144477352e-31  *Conclusion: Since the p-value is < 0.001, the correlation between highway-mpg and price is statistically significant, and the coefficient of ~ -0.705 shows that the relationship is negative and moderately strong.* |   ***ANOVA: Analysis of Variance***  *The Analysis of Variance (ANOVA) is a statistical method used to test whether there are significant differences between the means of two or more groups. ANOVA returns two parameters:*  *F-test score: ANOVA assumes the means of all groups are the same, calculates how much the actual means deviate from the assumption, and reports it as the F-test score. A larger score means there is a larger difference between the means.*  *P-value: P-value tells how statistically significant is our calculated score value.*  *If our price variable is strongly correlated with the variable, we are analyzing, expect ANOVA to return a sizeable F-test score and a small p-value.*  *Since ANOVA analyzes the difference between different groups of the same variable, the group by function will come in handy. Because the ANOVA algorithm averages the data automatically, we do not need to take the average beforehand.*  # Let's group the data first, to see if different types of 'drive-wheels' impacts 'price' or not  grouped\_test2=[df\_gptest](#df_gptest)[['drive-wheels', 'price']].groupby(['drive-wheels'])  grouped\_test2.head(2)   |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | id | drive-wheels | price | 1 | rwd | 16500.0 | 4 | 4wd | 17450.0 | 136 | 4wd | 7603.0 | | 0 | rwd | 13495.0 | 3 | fwd | 13950.0 | 5 | fwd | 15250.0 |  |  |  |   # Now obtain the values of the group using the method "get\_group".  grouped\_test2.get\_group('4wd')['price']  4 17450.0  136 7603.0  … …..  Name: price, dtype: float64  ANOVA for groups together 4wd, fwd and rwd:  # Use the function 'f\_oneway' in the module 'stats' to obtain the F-test score and P-value.  f\_val, p\_val = stats.f\_oneway(grouped\_test2.get\_group('fwd')['price'], grouped\_test2.get\_group('rwd')['price'], grouped\_test2.get\_group('4wd')['price'])  print ("ANOVA results: F=", f\_val, ", P =", p\_val)  ANOVA results: F= 67.95406500780399, P = 3.3945443577151245e-23  *This is a great result, with a large F test score showing a strong correlation and a P value of almost 0 implying almost certain statistical significance. But does this mean all three tested groups are all this highly correlated?*  ANOVA for groups separately:  # fwd and rwd group  f\_val, p\_val = stats.f\_oneway(grouped\_test2.get\_group('fwd')['price'], grouped\_test2.get\_group('rwd')['price'])  print ("ANOVA results: F=", f\_val, ", P =", p\_val )  ANOVA results: F= 130.5533160959111, P = 2.2355306355677845e-23  *Let's examine the other groups separately*  # 4wd and rwd  f\_val, p\_val = stats.f\_oneway(grouped\_test2.get\_group('4wd')['price'], grouped\_test2.get\_group('rwd')['price'])  print ("ANOVA results: F=", f\_val, ", P =", p\_val)  ANOVA results: F= 8.580681368924756, P = 0.004411492211225333  # 4wd and fwd  f\_val, p\_val = stats.f\_oneway(grouped\_test2.get\_group('4wd')['price'], grouped\_test2.get\_group('fwd')['price'])  print("ANOVA results: F=", f\_val, ", P =", p\_val)  ANOVA results: F= 0.665465750252303, P = 0.41620116697845666  *Conclusion: Important Variables*  *We now have a better idea of what our data looks like and which variables are important to take into account when predicting the car price. We have narrowed it down to the following variables:*  *Important continuous numerical variables are Length, Width, Curb-weight, Engine-size, Horsepower, City-mpg, Highway-mpg, Wheel-base and Bore. And important categorical variable is Drive-wheels.*  *As we now move into building machine learning models to automate our analysis, feeding the model with variables that meaningfully affect our target variable will improve our model's prediction performance.* |
| Model Development  *In this section, we will develop several models that will predict the price of the car using the variables or features. This is just an estimate but should give us an objective idea of how much the car should cost.*  *Some questions we want to ask in this module:*  *do I know if the dealer is offering fair value for my trade-in?*  *do I know if I put a fair value on my car?*  *Data Analytics, we often use Model Development to help us predict future observations from the data we have.*  *A Model will help us understand the exact relationship between different variables and how these variables are used to predict the result.*  # Setup: Import libraries  import pandas as pd  import numpy as np  import matplotlib.pyplot as plt  # Load data and store in data frame df:  path = 'https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DA0101EN/automobileEDA.csv'  df = pd.read\_csv(path)  df.head()  Linear Regression and Multiple Linear Regression:  *One example of a Data Model that we will be using is:*   * 1. *Simple Linear Regression (SLR): SLR is a method to help us understand the relationship between two variables: Where, X is the predictor/independent variable and Y is the response/dependent variable that we want to predict. The result of Linear Regression is a linear function that predicts the 𝑅𝑒𝑠𝑝𝑜𝑛𝑠𝑒 (dependent) 𝑉𝑎𝑟𝑖𝑎𝑏𝑙𝑒 𝑌 as a function of the 𝑃𝑟𝑒𝑑𝑖𝑐𝑡𝑜𝑟 (independent) 𝑉𝑎𝑟𝑖𝑎𝑏𝑙𝑒𝑠 𝑋.*   Linear function: 𝑌ℎ𝑎𝑡=𝑎+𝑏𝑋  𝑎 refers to the intercept of the regression line 0, in other words: the value of Y when X is 0  b refers to the slope of the regression line, meaning: the value with which Y changes when X increases by 1 unit  # Let’s load the modules for linear regression  from sklearn.linear\_model import LinearRegression  # Create the linear regression object  lm = LinearRegression()  lm  LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None, normalize=False)  # How could Highway-mpg help us predict car price?  *For example, we want to look at how highway-mpg can help us predict car price. Using simple linear regression, we will create a linear function with "highway-mpg" as the predictor and the "price" as the response variable*.  X = df[['highway-mpg']]  Y = df['price']  # Fit the linear model using highway-mpg.  lm.fit(X,Y)  # We can output a prediction  Yhat=lm.predict(X)  Yhat[0:5]  array([16236.50464347, 16236.50464347, 17058.23802179, 13771.3045085 , 20345.17153508])  # What is the value of the intercept (a)?  lm.intercept\_  38423.3058581574  # What is the value of the Slope (b)?  lm.coef\_  array([-821.73337832])  # What is the final estimated linear model we get?  As we saw above, we should get a final linear model with the structure:  # 𝑌ℎ𝑎𝑡=𝑎+𝑏𝑋, plugging in the actual values we get:  price = 38423.31 - 821.73 x highway-mpg  *Create a linear regression object and train the model to predict 'price' using 'engine-size'*  my\_lm = LinearRegression() *# Create Linear Regression object*  X = df[['engine-size']] *# Assigned variable to data*  Y = df['price']  my\_lm.fit(X,Y) *# Training the model*  my\_lm.coef\_ *# Find the slope*  array([166.86001569])  my\_lm.intercept\_ # Find the intercept  -7963.338906281042  Price = -7963.34 + 166.86\*engine-size *# What is the equation of the predicted line*  # What if we want to predict car price using more than one variable?  *If we want to use more variables in our model to predict car price, we can use Multiple Linear Regression.*   * 1. *Multiple Linear Regression (MLR)*: *MLR is very similar to Simple Linear Regression, but this method is used to explain the relationship between one continuous response (dependent) variable and two or more predictor (independent) variables. Most of the real-world regression models involve multiple predictors. We will illustrate the structure by using four predictor variables, but these results can generalize to any integer:*   The equation (Linear function) is 𝑌ℎ𝑎𝑡=𝑎+𝑏1𝑋1+𝑏2𝑋2+𝑏3𝑋3+𝑏4𝑋4   |  | | --- | | 𝑌: 𝑅𝑒𝑠𝑝𝑜𝑛𝑠𝑒 𝑉𝑎𝑟𝑖𝑎𝑏𝑙𝑒 | | 𝑋1: 𝑃𝑟𝑒𝑑𝑖𝑐𝑡𝑜𝑟 𝑉𝑎𝑟𝑖𝑎𝑏𝑙𝑒 1, 𝑋2: 𝑃𝑟𝑒𝑑𝑖𝑐𝑡𝑜𝑟 𝑉𝑎𝑟𝑖𝑎𝑏𝑙𝑒 2, 𝑋3: 𝑃𝑟𝑒𝑑𝑖𝑐𝑡𝑜𝑟 𝑉𝑎𝑟𝑖𝑎𝑏𝑙𝑒 3 …. | | 𝑎: 𝑖𝑛𝑡𝑒𝑟𝑐𝑒𝑝𝑡, in other words: the value of Y when X is 0 | | 𝑏1: 𝑐𝑜𝑒𝑓𝑓𝑖𝑐𝑖𝑒𝑛𝑡𝑠 𝑜𝑓 𝑉𝑎𝑟𝑖𝑎𝑏𝑙𝑒 1, 𝑏2: 𝑐𝑜𝑒𝑓𝑓𝑖𝑐𝑖𝑒𝑛𝑡𝑠 𝑜𝑓 𝑉𝑎𝑟𝑖𝑎𝑏𝑙𝑒 2, 𝑏3: 𝑐𝑜𝑒𝑓𝑓𝑖𝑐𝑖𝑒𝑛𝑡𝑠 𝑜𝑓 𝑉𝑎𝑟𝑖𝑎𝑏𝑙𝑒 3 …. |   # From the previous section we know that other good predictors of price could be:  Horsepower, Curb-weight, Engine-size, Highway-mpg  *# Let's develop a model using these variables as the predictor variables.*  Z = df[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']]  lm.fit(Z, df['price']) *# Fit the linear model using the four above-mentioned variables.*  lm.intercept\_ *#What is the value of the intercept(a)?*  -15806.62462632922  lm.coef*\_# What are the values of the coefficients (b1, b2, b3, b4)?*  array([53.49574423, 4.70770099, 81.53026382, 36.05748882])  # What is the final estimated linear model that we get:  𝑌ℎ𝑎𝑡=𝑎+𝑏1𝑋1+𝑏2𝑋2+𝑏3𝑋3+𝑏4𝑋4 # As we saw above, we should get a final linear function with the structure  # What is the linear function we get in this example?  Price = -15806.62 + 53.49 x horsepower + 4.70 x curb-weight + 81.53 x engine-size + 33.58258185 x highway-mpg  *Create and train a Multiple Linear Regression model "lm2" where the response variable is price, and the predictor variable is 'normalized-losses' and 'highway-mpg'.*  lm2 = LinearRegression()  lm2.fit(df[['normalized-losses', 'highway-mpg']], df['price'])  # Find the coefficient of the model?  Lm2.coef\_  print('coef is', lm2.coef\_,'and intercept is', lm2.intercept\_)  coef is [ 1.49789586 -820.45434016] and intercept is 38201.31327245728 |
| Model Evaluation using Visualization  *Now that we've developed some models, how do we evaluate our models and how do we choose the best one? One way to do this is by using visualization.*  # import the visualization package: seaborn  import seaborn as sns  %matplotlib inline    *Regression Plot*  *When it comes to simple linear regression, an excellent way to visualize the fit of our model is by using regression plots.*  *This plot will show a combination of a scattered data points (a scatter plot), as well as the fitted linear regression line going through the data. This will give us a reasonable estimate of the relationship between the two variables, the strength of the correlation, as well as the direction (positive or negative correlation).*  # Let's visualize Horsepower as potential predictor variable of price:  width = 12  height = 10  plt.figure(figsize=(width, height))  sns.regplot(x="highway-mpg", y="price", data=df)  plt.ylim(0,)  *We can see from this plot that price is negatively correlated to highway-mpg, since the regression slope is negative. One thing to keep in mind when looking at a regression plot is to pay attention to how scattered the data points are around the regression line. This will give you a good indication of the variance of the data, and whether a linear model would be the best fit or not. If the data is too far off from the line, this linear model might not be the best model for this data.*  # Let's compare this plot to the regression plot of "peak-rpm".  plt.figure(figsize=(width, height))  sns.regplot(x="peak-rpm", y="price", data=df)  plt.ylim(0,)  *Comparing the regression plot of "peak-rpm" and "highway-mpg" we see that the points for "highway-mpg" are much closer to the generated line and on the average decrease. The points for "peak-rpm" have more spread around the predicted line, and it is much harder to determine if the points are decreasing or increasing as the "highway-mpg" increases.*  # Given the regression plots above is "peak-rpm" or "highway-mpg" more strongly correlated with "price”?  # Use the method. corr () to verify your answer.  df[['peak-rpm','highway-mpg','price']].corr()   |  |  |  |  | | --- | --- | --- | --- | |  | peak-rpm | highway-mpg | price | | peak-rpm | 1.000000 | -0.058598 | -0.101616 | | highway-mpg | -0.058598 | 1.000000 | -0.704692 | | price | -0.101616 | -0.704692 | 1.000000 |   The variable "highway-mpg" has a stronger correlation with "price", it is approximate -0.704692 compared to "peak-rpm" which is approximate -0.101616.  *Residual Plot*  A good way to visualize the variance of the data is to use a residual plot.  What is a residual?  *The difference between the observed value (y) and the predicted value (Yhat) is called the residual (e). When we look at a regression plot, the residual is the distance from the data point to the fitted regression line.*  So, A residual plot *is a graph that shows the residuals on the vertical y-axis and the independent variable on the horizontal x-axis.*  What do we pay attention to when looking at a residual plot?  *We look at the spread of the residuals: If the points in a residual plot are randomly spread out around the x-axis, then a linear model is appropriate for the data.*  *Why is that? Randomly spread out residuals means that the variance is constant, and thus the linear model is a good fit for this data.*  width = 12  height = 10  plt.figure(figsize=(width, height))  sns.residplot(df['highway-mpg'], df['price'])  plt.show()  What is this plot telling us?  *We can see from this residual plot that the residuals are not randomly spread around the x-axis, which leads us to believe that maybe a non-linear model is more appropriate for this data.*  *Distribution Plot*  How do we visualize a model for Multiple Linear Regression? *This gets a bit more complicated because you can't visualize it with regression or residual plot.*  One way to look at the fit of the model is by looking at the distribution plot: *We can look at the distribution of the fitted values that result from the model and compare it to the distribution of the actual values.*  # *Let’s create Distribution Plot for Multiple Linear Regression*  # First let’s make a prediction  Y\_hat = lm.predict(Z)  #Create plot using Seaborn function “.distplot()”  plt.figure(figsize=(width, height))  ​  ax1 = sns.distplot(df['price'], hist=False, color="r", label="Actual Value")  sns.distplot(Yhat, hist=False, color="b", label="Fitted Values" , ax=ax1)  ​  ​plt.title('Actual vs Fitted Values for Price')  plt.xlabel('Price (in dollars)')  plt.ylabel('Proportion of Cars')  ​  plt.show()  plt.close()  We can see that the fitted values are reasonably close to the actual values, since the two distributions overlap a bit. However, there is definitely some room for improvement. |
| *Multivariate Polynomial function*  The analytical expression for Multivariate Polynomial function gets complicated. For example, the expression for a second-order (degree=2) polynomial with two variables (x1 and x2) is given by:  𝑌ℎ𝑎𝑡=𝑎+𝑏1𝑋1+𝑏2𝑋2+𝑏3𝑋1𝑋2+𝑏4+𝑏5  # We can perform a polynomial transform on multiple features. First, we import the module:  from sklearn.preprocessing import PolynomialFeatures  # We create a PolynomialFeatures object of degree 2:  pr=PolynomialFeatures(degree=2)  pr  PolynomialFeatures(degree=2, include\_bias=True, interaction\_only=False)  Z\_pr=pr.fit\_transform(Z)  # The original data is of 201 samples and 4 features  Z.shape  (201, 4)  # After the transformation, there 201 samples and 15 features  Z\_pr.shape  (201, 15)  *Pipeline*:  *Data Pipelines simplify the steps of processing the data.*  # We use the module Pipeline to create a pipeline. We also use StandardScaler as a step in our pipeline.  from sklearn.pipeline import Pipeline  from sklearn.preprocessing import StandardScaler  # We create the pipeline, by creating a list of tuples including the name of the model or estimator and its corresponding constructor.  Input=[('scale',StandardScaler()), ('polynomial', PolynomialFeatures(include\_bias=False)), ('model',LinearRegression())]  # We input the list as an argument to the pipeline constructor  pipe=Pipeline(Input)  pipe  Pipeline(memory=None,  steps=[('scale', StandardScaler(copy=True, with\_mean=True, with\_std=True)), ('model', LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None, normalize=False))])  # We can normalize the data, perform a transform and fit the model simultaneously.  pipe.fit(Z,y)  # Similarly, we can normalize the data, perform a transform and produce a prediction simultaneously  ypipe=pipe.predict(Z)  ypipe[0:4]  array([13699.11161184, 13699.11161184, 19051.65470233, 10620.36193015])  # Create a pipeline that Standardizes the data, then perform prediction using a linear regression model using the features Z and targets y  from sklearn.pipeline import Pipeline  from sklearn.preprocessing import StandardScaler  Input=[('scale',StandardScaler()), ('model',LinearRegression())]  pipe = Pipeline(Input)  pipe.fit(Z,y)  ypipe = pipe.predict(Z)  ypipe[0:10]  array([13699.11161184, 13699.11161184, 19051.65470233, 10620.36193015,  15521.31420211, 13869.66673213, 15456.16196732, 15974.00907672,  17612.35917161, 10722.32509097])  *Measures for In-Sample Evaluation*  *When evaluating our models, not only do we want to visualize the results, but we also want a quantitative measure to determine how accurate the model is. Two very important measures that are often used in Statistics to determine the accuracy of a model are*:   * R^2 / R-squared: *R squared, also known as the coefficient of determination, is a measure to indicate how close the data is to the fitted regression line. The value of the R-squared is the percentage of variation of the response variable (y) that is explained by a linear model.* *Higher R-squared value is a better fit for the data.* * Mean Squared Error (MSE): *The Mean Squared Error measures the average of the squares of errors, that is, the difference between actual value (y) and the estimated value (ŷ).* *Smallest MSE value is a better fit for the data.*   Model 1: Simple Linear Regression  # Let's calculate the R^2 for highway\_mpg\_fit  # Fit the model  lm.fit(X, Y)  # Find the R^2  print ('The R-square is: ', lm.score(X, Y))  The R-square is: 0.4965911884339176  *We can say that ~ 49.659% of the variation of the price is explained by this simple linear model " highway\_mpg\_fit".*  # Let's calculate the MSE for highway\_mpg\_fit  We can predict the output i.e., "yhat" using the predict method, where X is the input variable:  Yhat=lm.predict(X)  print('The output of the first three predicted value is: ', Yhat[0:3])  The output of the first four predicted value is: [16236.50464347 16236.50464347 17058.23802179]  # Lets import the function mean\_squared\_error from the module metrics  from sklearn.metrics import mean\_squared\_error  # We compare the predicted results with the actual results  mse = mean\_squared\_error(df['price'], Yhat)  print('The mean square error of price and predicted value is: ', mse)  The mean square error of price and predicted value is: 31635042.944639888  Model 2: Multiple Linear Regression  # Let's calculate the R^2 for MSR with Z having multiple variable inputs  # Fit the model  lm.fit(Z, df['price'])  print('The R-square is: ', lm.score(Z, df['price']))  The R-square is: 0.8093562806577457  *We can say that ~ 80.896 % of the variation of price is explained by this multiple linear regression "multi\_fit".*  # Let's calculate the MSE  # We produce a prediction  Y\_predict\_multifit = lm.predict(Z)  # We compare the predicted results with the actual results  print('The mean square error of price and predicted value using multi\_fit is: ', \  mean\_squared\_error(df['price'], Y\_predict\_multifit))  The mean square error of price and predicted value using multi\_fit is: 11980366.87072649  Model 3: Polynomial Fit  # Let's calculate the R^2  # let’s import the function r2\_score from the module metrics as we are using a different function (p())  from sklearn.metrics import r2\_score  # We apply the function to get the value of r^2  r\_squared = r2\_score(y, p(x))  print('The R-square value is: ', r\_squared)  The R-square value is: 0.674194666390652  *We can say that ~ 67.419 % of the variation of price is explained by this polynomial fit*  # Let's calculate the MSE  mean\_squared\_error(df['price'], p(x))  20474146.426361218 |
| *Prediction and Decision Making*  Prediction: *In the previous section, we trained the model using the method fit. Now we will use the method predict*  # Let’s import pyplot for plotting; we will also be using some functions from numpy.  import matplotlib.pyplot as plt  import numpy as np  ​%matplotlib inline  new\_input=np.arange(1, 100, 1).reshape(-1, 1) # Create a new input  lm.fit(X, Y) # Fit the model  yhat=lm.predict(new\_input) # Produce a prediction  yhat[0:5]  array([37601.57247984, 36779.83910151, 35958.10572319, 35136.37234487, 34314.63896655])  plt.plot(new\_input, yhat) # Plot the data  plt.show()  Decision Making or Determining a Good Model Fit:  *Now that we have visualized the different models, and generated the R-squared and MSE values for the fits, how do we determine a good model fit?*  *When comparing models, the model with the higher R-squared and the smallest MSE value is a better fit for the data.*  # Let's take a look at the values for the different models.   |  |  |  | | --- | --- | --- | | Model | R-squared | MSE | | SLR: Using Highway-mpg as a predictor variable of Price. | 0.49659118843391759 | 3.16 x10^7 | | MLR: Using Horsepower, Curb-weight, Engine-size, and Highway-mpg | *0.80896354913783497* | *1.2 x10^7* | | Polynomial Fit: Using Highway-mpg as a predictor variable of Price. | 0.6741946663906514 | 2.05 x 10^7 |  1. Simple Linear Regression model (SLR) vs Multiple Linear Regression model (MLR)   *Usually, the more variables you have, the better your model is at predicting, but this is not always true. Sometimes you may not have enough data, you may run into numerical problems, or many of the variables may not be useful and or even act as noise. As a result, you should always check the MSE and R^2.*  *So, to be able to compare the results of the MLR vs SLR models, we look at a combination of both the R-squared and MSE to make the best conclusion about the fit of the model.*  MSE: The MSE of SLR is 3.16x10^7 while MLR has an MSE of 1.2 x10^7. The MSE of MLR is much smaller.  R-squared: In this case, we can also see that there is a big difference between the R-squared of the SLR and the R-squared of the MLR. The R-squared for the SLR (~0.497) is very small compared to the R-squared for the MLR (~0.809).  *This R-squared along with the MSE show that MLR seems like the better model fit in this case, compared to SLR.*   1. Simple Linear Model (SLR) vs Polynomial Fit:   MSE: We can see that Polynomial Fit brought down the MSE, since this MSE is smaller than the one from the SLR.  R-squared: The R-squared for the Polyfit is larger than the R-squared for the SLR, so the Polynomial Fit also brought up the R-squared quite a bit.  *Since the Polynomial Fit resulted in a lower MSE and a higher R-squared, we can conclude that this was a better fit model than the simple linear regression for predicting Price with Highway-mpg as a predictor variable.*   1. Multiple Linear Regression (MLR) vs Polynomial Fit   MSE: The MSE for the MLR is smaller than the MSE for the Polynomial Fit.  R-squared: The R-squared for the MLR is also much larger than for the Polynomial Fit.  **Conclusion**: Comparing these three models, we conclude that the MLR model is the best model to be able to predict price from our dataset. This result makes sense, since we have 27 variables in total, and we know that more than one of those variables are potential predictors of the final car price. |
| Model Evaluation and Refinement  *We have built models and made predictions of car prices. Now we will determine how accurate these predictions are.*  *First, Let’s setup the environment for model evaluation*  # Import important libraries  import pandas as pd  import numpy as np  ​# Import clean data  path = 'https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DA0101EN/module\_5\_auto.csv'  df = pd.read\_csv(path)  df.to\_csv('module\_5\_auto.csv')  # only use numeric data first  df=df.\_get\_numeric\_data()  df.head()  # Libraries for plotting  %%capture  ! pip install ipywidgets  from IPython.display import display  from IPython.html import widgets  from IPython.display import display  from ipywidgets import interact, interactive, fixed, interact\_manual  # Function#1 for plotting  def DistributionPlot(RedFunction, BlueFunction, RedName, BlueName, Title):  width = 8  height = 6  plt.figure(figsize=(width, height))  ​ ax1 = sns.distplot(RedFunction, hist=False, color="r", label=RedName)  ax2 = sns.distplot(BlueFunction, hist=False, color="b", label=BlueName, ax=ax1)  ​ plt.title(Title)  plt.xlabel('Price (in dollars)')  plt.ylabel('Proportion of Cars')  ​ plt.show()  plt.close()  # Function#2 for plotting  def PollyPlot(xtrain, xtest, y\_train, y\_test, lr,poly\_transform):  width = 8  height = 6  plt.figure(figsize=(width, height))  # training data  # testing data  # lr: linear regression object  # poly\_transform: polynomial transformation object  xmax=max([xtrain.values.max(), xtest.values.max()])  ​ xmin=min([xtrain.values.min(), xtest.values.min()])  ​ x=np.arange(xmin, xmax, 0.1)  ​ plt.plot(xtrain, y\_train, 'ro', label='Training Data')  plt.plot(xtest, y\_test, 'go', label='Test Data')  plt.plot(x, lr.predict(poly\_transform.fit\_transform(x.reshape(-1, 1))), label='Predicted Function')  plt.ylim([-10000, 60000])  plt.ylabel('Price')  plt.legend()  ***Model Evaluation***  Part 1: Training and Testing  *An important step in testing your model is to split your data into training and testing data. We will place the predicter variables(X) and target data variable price(Y) in a separate dataframe x and y.*  y\_data = df['price'] *# Create list of ‘price’ data* named y\_data  x\_data = df.drop('price',axis=1) # Drop price column from dataframe(df) and create copy as x\_data  *Now, we randomly split our data into training and testing data using the function train\_test\_split.*  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=0.15, random\_state=1)  print("number of test samples:", x\_test.shape[0])  print("number of training samples:",x\_train.shape[0])  ​  number of test samples: 31  number of training samples: 170  *The test\_size parameter sets the proportion of data that is split into the testing set. In the above, the testing 15% data*  # Use the function "train\_test\_split" to split up the data set such that 40% of the data samples will be utilized for testing, set the parameter "random\_state" equal to zero.  from sklearn.model\_selection import train\_test\_split  x\_train\_1, x\_test\_1, y\_train\_1, y\_test\_1 = train\_test\_split(x\_data, y\_data, test\_size=0.4, random\_state=0)  print("number of test samples:", x\_test\_1.shape[0])  print("number of training samples:",x\_train\_1.shape[0])  number of test samples: 81  number of training samples: 120  # Let's create linear regression object and fit model  from sklearn.linear\_model import LinearRegression *# Import module*  lre=LinearRegression() *# Create a Linear Regression object*  lre.fit(x\_train[['horsepower']], y\_train) *# Fit the model using the feature horsepower*  lre LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None,normalize=False)  lre.score(x\_test[['horsepower']], y\_test) *# Calculate the R^2 on the test data*  0.707688374146705  lre.score(x\_train[['horsepower']], y\_train)  0.6449517437659684 *# we can see the R^2 is much smaller using the test data*  # Find the R^2 on the test data using 90% of the data for training data  x\_train1, x\_test1, y\_train1, y\_test1 = train\_test\_split(x\_data, y\_data, test\_size=0.9, random\_state=0)  lre.fit(x\_train1[['horsepower']],y\_train1)  lre.score(x\_test1[['horsepower']],y\_test1)  0.6559543699796797  Cross-validation Score: *Sometimes you do not have sufficient testing data; as a result, you may want to perform Cross-validation. Let's go over several methods that you can use for Cross-validation.*  # Let’s import model\_selection from the module cross\_val\_score.  from sklearn.model\_selection import cross\_val\_score  *We input the object, the feature in this case ' horsepower', the target data (y\_data). The parameter 'cv' determines the number of folds; in this case 4.*  Rcross = cross\_val\_score(lre, x\_data[['horsepower']], y\_data, cv=4)  *The default scoring is R^2; each element in the array has the average R^2 value in the fold:*  Rcross array([0.7746232 , 0.51716687, 0.74785353, 0.04839605])  *We can calculate the average and standard deviation of our estimate:*  print("The mean of the folds are", Rcross.mean(), "and the standard deviation is" , Rcross.std())  *The means of the folds are 0.522009915042119 and the standard deviation is 0.291183944475603*  *We can use negative squared error as a score by setting the parameter 'scoring' metric to 'neg\_mean\_squared\_error'.*  -1 \* cross\_val\_score(lre,x\_data[['horsepower']], y\_data, cv=4,scoring='neg\_mean\_squared\_error')  array([20254142.84026703, 43745493.2650517 , 12539630.34014931, 17561927.72247591])  # Calculate the average R^2 for two folds, find the average R^2 for the second fold utilize the horsepower as a feature  Rc = cross\_val\_score(lre,x\_data[['horsepower']], y\_data,cv=2)  Rc.mean() # array([0.59015621, 0.44319613]).mean()  0.5166761697127429  Cross-validation Predict:  *You can also use the function 'cross\_val\_predict' to predict the output. The function splits up the data into the specified number of folds, using one-fold to get a prediction while the rest of the folds are used as test data.*  *#* First import the function  from sklearn.model\_selection import cross\_val\_predict  *We input the object, the feature in this case 'horsepower’, the target data y\_data. The parameter 'cv' determines the number of folds; in this case 4. We can produce an output*  yhat = cross\_val\_predict(lre,x\_data[['horsepower']], y\_data, cv=4)  yhat[0:5]  array ([14141.63807508, 14141.63807508, 20814.29423473, 12745.03562306, 14762.35027598])  ***Over-fitting, Under-fitting and Model Selection***  Part 2: Overfitting, Underfitting and Model Selection  *It turns out that the test data sometimes referred to as the out of sample data is a much better measure of how well your model performs in the real world. One reason for this is overfitting; let's go over some examples. It turns out these differences are more apparent in Multiple Linear Regression and Polynomial Regression so we will explore overfitting in that context.*  # Let's create Multiple linear regression objects and train the model using 'horsepower', 'curb-weight', 'engine-size' and 'highway-mpg' as features.  lr = LinearRegression()  lr.fit(x\_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y\_train)  LinearRegression(copy\_X=True, fit\_intercept=True, n\_jobs=None, normalize=False)  # Prediction using training data:  yhat\_train = lr.predict(x\_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])  yhat\_train[0:5]  # Prediction using test data:  yhat\_test = lr.predict(x\_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])  yhat\_test[0:5]  # Let's perform some model evaluation using our training and testing data separately.  import matplotlib.pyplot as plt  %matplotlib inline  import seaborn as sns  # Let's examine the distribution of the predicted values of the training data.  Title = 'Distribution Plot for Predicted Vs Original Value of Training Data  DistributionPlot(y\_train, yhat\_train, "Actual V(Train)", "Predicted V(Train)", Title)  Figure 1: Predicted values using the training data Vs. to the training data.  *So far, the model seems to be doing well in learning from the training dataset. But what happens when the model encounters new data from the testing dataset? When the model generates new values from the test data, we see the distribution of the predicted values is much different from the actual target values.*  Title='Distribution Plot for Predicted Value of Test Data vs Original Test Data'  DistributionPlot(y\_test,yhat\_test,"Actual V(Test)","Predicted V(Test)",Title)  Figure 2: Plot of predicted value using the test data Vs. to the test data.  Comparing Figure 1 and Figure 2; it is evident the distribution of the test data in Figure 1 is much better at fitting the data. This difference in Figure 2 is apparent where the ranges are from 5000 to 15,000. This is where the distribution shape is exceptionally different.  Let's see if polynomial regression also exhibits a drop in the prediction accuracy when analysing the test dataset.  # Let's try polynomial regression  from sklearn.preprocessing import PolynomialFeatures *# Import function*  Overfitting  *Overfitting occurs when the model fits the noise, not the underlying process. Therefore, when testing your model using the test-set, your model does not perform as well as it is modelling noise, not the underlying process that generated the relationship.*  # Let's create a degree 5 polynomial model, use 55 percent of the data for testing and the rest for training:  x\_train, x\_test, y\_train, y\_test = train\_test\_split(x\_data, y\_data, test\_size=0.45, random\_state=0) # *Split Data*  # Degree 5 polynomial transformation on the feature 'horse power'  pr = PolynomialFeatures(degree=5)  x\_train\_pr = pr.fit\_transform(x\_train[['horsepower']])  x\_test\_pr = pr.fit\_transform(x\_test[['horsepower']])  # Now let's create a linear regression model "poly" and train it.  poly = LinearRegression()  poly.fit(x\_train\_pr, y\_train)  *We can see the output of our model using the method "predict." then assign the values to "yhat".*  yhat = poly.predict(x\_test\_pr)  # Let's take the first five predicted values and compare it to the actual targets.  print("Predicted values:", yhat[0:4])  print("True values:", y\_test[0:4].values)  Predicted values: [ 6728.65584216 7307.98804276 12213.78788015 18893.2476361 ]  True values: [ 6295. 10698. 13860. 13499.]  *We will use the function "PollyPlot" that we defined at the beginning of the lab to display the training data, testing data, and the predicted function.*  PollyPlot(x\_train[['horsepower']], x\_test[['horsepower']], y\_train, y\_test, poly,pr)  *Figure on right, Polynomial regression model, red dots represent training data, green dots represent test data, and the blue line represents the model prediction.*  *We see that the estimated function appears to track the data but around 200 horsepower, the function begins to diverge from the data points.*  # R^2 of the training data:  poly.score(x\_train\_pr, y\_train)  0.5567716902126982  # R^2 of the test data:  poly.score(x\_test\_pr, y\_test)  -29.871341600352796  *We see the R^2 for the training data is 0.5567 while the R^2 on the test data was -29.87. The lower the R^2, the worse the model, a Negative R^2 is a sign of overfitting.*  *Let's see how the R^2 changes on the test data for different order polynomials and plot the results:*    Rsqu\_test = []  ​order = [1, 2, 3, 4]  for n in order:  pr = PolynomialFeatures(degree=n)  x\_train\_pr = pr.fit\_transform(x\_train[['horsepower']])  x\_test\_pr = pr.fit\_transform(x\_test[['horsepower']])  lr.fit(x\_train\_pr, y\_train)  Rsqu\_test.append(lr.score(x\_test\_pr, y\_test))  ​plt.plot(order, Rsqu\_test)  plt.xlabel('order')  plt.ylabel('R^2')  plt.title('R^2 Using Test Data')  plt.text(3, 0.75, 'Maximum R^2 ')  Text(3, 0.75, 'Maximum R^2 ')  *We see the R^2 gradually increases until an order three polynomial is used. Then the R^2 dramatically decreases at 4.*  *The following function “f” will be used in the next section; please run the cell.*  def f(order, test\_data):  x\_train, x\_test, y\_train, y\_test = train\_test\_split(x\_data, y\_data, test\_size=test\_data, random\_state=0)  pr = PolynomialFeatures(degree=order)  x\_train\_pr = pr.fit\_transform(x\_train[['horsepower']])  x\_test\_pr = pr.fit\_transform(x\_test[['horsepower']])  poly = LinearRegression()  poly.fit(x\_train\_pr,y\_train)  PollyPlot(x\_train[['horsepower']], x\_test[['horsepower']], y\_train,y\_test, poly, pr)  *The following interface allows you to experiment with different polynomial orders and different amounts of data.*  interact(f, order=(0, 6, 1), test\_data=(0.05, 0.95, 0.05))    *We can perform polynomial transformations with more than one feature., let’s create one and plot it.*  # Create a "PolynomialFeatures" object "pr1" of degree two.  pr1=PolynomialFeatures(degree=2)  # Transform the training & testing samples for the feature’s 'horsepower', 'curb-weight', 'engine-size' & 'highway-mpg'. Hint: Use the method "fit transform"  x\_train\_pr1 = pr1.fit\_transform(x\_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])  x\_test\_pr1 = pr1.fit\_transform(x\_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])  # How many dimensions does the new feature have? Hint: Use the attribute "shape"  x\_train\_pr1.shape  # Create a linear regression model "poly1" and train the object using the method "fit" using the polynomial features.  poly1 = LinearRegression().fit(x\_train\_pr1,y\_train)  # Use the method "predict" to predict an output on the polynomial features, then use the function "DistributionPlot" to display the distribution of the predicted output vs the test data.  yhat\_test1=poly1.predict(x\_test\_pr1)  Title='Predicted Value of Test Data vs Original Value of Test Data'  DistributionPlot(y\_test, yhat\_test1, "Act Val", "Predi Val", Title)  # Use the distribution plot to determine the two regions were the predicted prices are less accurate than the actual prices.  *"The predicted value is lower than actual value for cars where the price $10,000 range, conversely the predicted price is larger than the price cost in the $30,000 to $40,000 range. As such the model is not as accurate in these ranges."*  ***Ridge Regression***  *In this section, we will review Ridge Regression we will see how the parameter Alfa changes the model. Just a note here our test data will be used as validation data.*  # Let's perform a degree two polynomial transformation on our data.  pr=PolynomialFeatures(degree=2)  x\_train\_pr=pr.fit\_transform(x\_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg','normalized-losses','symboling']])  x\_test\_pr=pr.fit\_transform(x\_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg','normalized-losses','symboling']])  # Let's import Ridge from the module linear models.  from sklearn.linear\_model import Ridge  # Let's create a Ridge regression object, setting the regularization parameter to 0.1  RigeModel=Ridge(alpha=0.1)  # Like regular regression, you can fit the model using the method fit.  RigeModel.fit(x\_train\_pr, y\_train)  Ridge(alpha=0.1, copy\_X=True, fit\_intercept=True, max\_iter=None, normalize=False, random\_state=None, solver='auto', tol=0.001)  # Similarly, you can obtain a prediction:  yhat = RigeModel.predict(x\_test\_pr)  # Let's compare the first five predicted samples to our test set  print('predicted:', yhat[0:4])  print('test set:', y\_test[0:4].values)  predicted: [ 6567.83081933 9597.97151399 20836.22326843 19347.69543463]  test set: [ 6295. 10698. 13860. 13499.]  *We select the value of Alfa that minimizes the test error, for example, we can use a for loop.*  Rsqu\_test = []  Rsqu\_train = []  dummy1 = []  ALFA = 10 \* np.array(range(0,100))  for alfa in ALFA:  RigeModel = Ridge(alpha=alfa)  RigeModel.fit(x\_train\_pr, y\_train)  Rsqu\_test.append(RigeModel.score(x\_test\_pr, y\_test))  Rsqu\_train.append(RigeModel.score(x\_train\_pr, y\_train))  *We can plot out the value of R^2 for different Alphas*  width = 8  height = 12  plt.figure(figsize=(width, height))  ​plt.plot(ALFA,Rsqu\_test, label='validation data ')  plt.plot(ALFA,Rsqu\_train, 'r', label='training Data ')  plt.xlabel('alpha')  plt.ylabel('R^2')  plt.legend()  *The blue line in Figure represents the R^2 of the test data, and the red line represents the R^2 of the training data. The x-axis represents the different values of Alfa*  *The red line in represents the R^2 of the training data, as Alpha increases the R^2 decreases; therefore, as Alfa increases the model performs worse on the training data.*  *Perform Ridge regression and calculate the R^2 using the polynomial features, use the training data to train the model and test data to test the model. The parameter alpha should be set to 10.*  RigeModel=Ridge(alpha=10)  RigeModel.fit(x\_train\_pr, y\_train)  RigeModel.score(x\_test\_pr, y\_test)  ***Grid Search***  *The term Alfa is a hyperparameter, sklearn has the class GridSearchCV to make the process of finding the best hyperparameter simpler.*  # Let's import GridSearchCV from the module model\_selection.  from sklearn.model\_selection import GridSearchCV  # We create a dictionary of parameter values:  parameters1= [{'alpha': [0.001,0.1,1, 10, 100, 1000, 10000, 100000, 100000]}]  # Create a ridge regions object:  RR=Ridge()  RR  Ridge(alpha=1.0, copy\_X=True, fit\_intercept=True, max\_iter=None, normalize=False, random\_state=None, solver='auto', tol=0.001)  # Create a ridge grid search object  Grid1 = GridSearchCV(RR, parameters1,cv=4)  # Fit the model  Grid1.fit(x\_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y\_data)  GridSearchCV(cv=4, error\_score='raise-deprecating', estimator=Ridge(alpha=1.0, copy\_X=True, fit\_intercept=True, max\_iter=None, normalize=False, random\_state=None, solver='auto', tol=0.001), fit\_params=None, iid='warn', n\_jobs=None, param\_grid=[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 100000]}], pre\_dispatch='2\*n\_jobs', refit=True, return\_train\_score='warn', scoring=None, verbose=0)  *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 BestRR as follows.*  BestRR=Grid1.best\_estimator\_  BestRR  Ridge(alpha=10000, copy\_X=True, fit\_intercept=True, max\_iter=None, normalize=False , random\_state=None, solver='auto', tol=0.001)  # We now test our model on the test data  BestRR.score(x\_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y\_test)  0.8411649831036152  *Perform a grid search for the alpha parameter and the normalization parameter, then find the best values of the parameters*  parameters2= [{'alpha': [0.001,0.1,1, 10, 100, 1000,10000,100000,100000],'normalize':[True,False]} ]  Grid2 = GridSearchCV(Ridge(), parameters2,cv=4)  Grid2.fit(x\_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],y\_data)  Grid2.best\_estimator\_  Ridge(alpha=0.1, copy\_X=True, fit\_intercept=True, max\_iter=None, normalize=True, random\_state=None, solver='auto', tol=0.001)  Assignment Notebook Link: <https://cocl.us/da0101en_coursera_labb> |
|  |