Welcome to your 310 Portfolio

A progression of learning for CSC/DSP 310: Programming for Data Science at University of Rhode Island.

About Me

Hello, my name is Brianna MacDonald and I am a Senior at URI with a double major in Computer Engineering and Chinese with a double minor in Cyber Security and Data Science. I've been studying Chinese for about 4 years and I recently had the oppertunity to go to Shanghai and Beijing last Winter right before COVID-19.

Data Science, to me

Data Science is the intersection between computer science, statistics, and domain knowledge. Data Science has many different uses, such as in medical sciences or in machine learning.

There are many different components of Data Science. The four main components of Data Science are Data Strategy, Data Engineering, Data Analysis and Modeling, and Data Visualization/Operationalization. Data Strategy is about determing what data you want to gather and why. It makes a connection between the data you want to gather and the goals for that data. Data Engineering is about the systems and technology that are used to leverage, access, organize, and use the data. Data Analysis/Modeling is about describing or predicting data, creating analysis and assumptions about data, and mathematically modeling the data. Data Visualization/Operationalization is about visualizing data and understanding how different visuals describe the data, as well as making the data operational by making a machine/person make a decision or action based on the computing of the data.

Compute the Grade for CSC/DSP 310

- To run by entering values into function, please run **compute_grade()** with desired values.

```
def compute_grade(num_level1, num_level2, num_level3):
    """
    Computes a grade for CSC/DSP 310 from numbers of achievements earned at each level
    :param num_level1: int, number of level 1 achievements earned
    :param num_level2: int, number of level 2 achievements earned
    :param num_level3: int, number of level 3 achievements earned
    :return: letter_grade: string, letter grade with possible modifier (+/-)
    """
    # Initializing Variables
    letter_grade = ""
    total_grade = num_level1 + num_level2 + num_level3

    # Error Handling
    if total_grade > 45:
        print("Invalid total. Please re-enter values.")

    # Definitions of Grades
    else:
        if 3 <= num_level1 < 5:
            letter_grade = 'D'
        elif 5 <= num_level1 < 10:
            letter_grade = 'D+'
        elif 10 <= num_level1 < 15:
            letter_grade = 'C-'
        elif num_level1 == 15 and 0 <= num_level2 < 5:
            letter_grade = 'C'
        elif num_level1 == 15 and 5 <= num_level2 < 10:
            letter_grade = 'C+'
        elif num_level1 == 15 and 10 <= num_level2 < 15:
            letter_grade = 'B-'
        elif num_level1 == 15 and num_level2 == 15 and 0 <= num_level3 < 5:
            letter_grade = 'B'
        elif num_level1 == 15 and num_level2 == 15 and 5 <= num_level3 < 10:
            letter_grade = 'B+'
        elif num_level1 == 15 and num_level2 == 15 and 10 <= num_level3 < 15:
            letter_grade = 'A-'
        elif num_level1 == 15 and num_level2 == 15 and num_level3 == 15:
            letter_grade = 'A'
        else:
            print("Does not translate to letter grade.")
    print(f'Your grade is {letter_grade}.')
```

The example below will give a grade of a C.

```
# Example 1
compute_grade(15, 2, 0)
```

The example below will give a grade of a B.

```
# Example 2
compute_grade(15, 15, 2)
```

The example below will give a grade of an A-.

```
# Example 3
compute_grade(15, 15, 12)
```

Portfolio Check 1

I've demonstrated skills *access*, *construct*, and *prepare* in chapter 1. This file can be found here.
I've demonstrated skill *process* in chapter 2. The file can be found here.
I've demonstrated skills *summarize*, *construct*, and *visualize* in chapter 3. The file can be found here.
I've demonstrated skills *prepare*, *summarize*, and *visualize* in chapter 4. The file can be found here.
All the files linked above are **.md** files, however you can also view the **.ipynb** files by clicking through the chapters.

Throughout all chapters, I believe I have earned *python level 3* as I have used and created reliable and efficient code that consistently adheres to pep8.

Submission 1 - Chapter 1

In this section of my portfolio I will be demonstrating skills for **python level 3, prepare level 2, construct level 3, and access level 3**. For more demonstration on **prepare**, please look here.

Access Level 3

Access data from both common and uncommon formats and identify best practices for formats in different contexts.

There are many different ways to access data in Pandas, whether you want to read into a more common format such as a CSV file, or if you to access data stored in an HDF5 table or in SAS format.

When considering which format to use, you should also note the different types of separators that are the defaults of each method. For example, pd.read_csv defaults separation to a comma, while pd.read_table defaults to'\t'. Different file types also come with different parameters (whether optional or required) according to their designated Pandas function.

For a contextual example, consider pd.read_sql_table, pd.read_sql_query, and pd.read_sql. Although these functions seem very similar to each other, there are actually quite a few key differences. The list below describes each method and how you can use them.

pd.read_sql_table

- Reads an SQL database table into a DataFrame
- Takes in parse_dates, columns, chunksize
- Does not support DBAPI connections

pd.read_sql_query

- Reads an SQL query into a DataFrame
- Takes in parse_dates, chunksize
- May support DBAPI connections depending on type

pd.read_sql

- Read SQL query or database table into a DataFrame
- More for convenience, compatibility for previous methods
- Takes in parse_dates, columns, chunksize

Construct Level 3

Now, I will be using SQLite to access the database for a dataset called **Simple Folk**.

Link to database: http://2016.padjo.org/files/data/starterpack/simplefolks.sqlite

```
import sqlite3
import pandas as pd
con = sqlite3.connect('simplefolks.sqlite')
cur = con.cursor()
rows = cur.fetchall()
```

```
# DataFrame with the age, sex, and name of individuals
people_df = pd.read_sql_query("SELECT age, sex, name FROM people", con)
# DataFrame with pets, pet owners, and pet names
pet_df = pd.read_sql_query("SELECT name, owner_name FROM pets ORDER BY name", con)
```

```
# Viewing DataFrames
people_df.head()
```

```
# Viewing DataFrames
pet_df.head()
```

Now, let's say we wanted to list all of the 30 year old and older men in the people table. We can do this by implementing the query below.

```
query_1 = pd.read_sql_query("SELECT * FROM people WHERE sex = 'M' AND age >= 30", con)
query_1
```

Or, with the pet table we can find the pets name and type that are not dogs or cats. Personally, the bird named Harambe is my favorite.

```
query_2 = pd.read_sql_query("SELECT name, type FROM pets WHERE type != 'cat' and TYPE != 'dog'",
con)
query_2
```

Luckily for us, SQLite databases and their respective tables usually come in a very easy-to-read format. In other cases where the data is not as easy to read in, we need to be able to clean and prepare it.

Prepare Level 2

Apply data reshaping, cleaning, and filtering as directed.

```
# First, I'll load the xls file in and take a look at it uncleaned.
unclean_table = pd.read_excel("tabn039.xls")
unclean_table.head()
```

Obviously, this table needs some major cleaning. Although I am only showing the head of the table above, there are actually plenty of rows towards the end (rows 42-45) that are filled with NaN values. Our first task will be to **drop and rename** columns and rows.

```
clean_table = pd.read_excel("tabn039.xls", header = 1, index_col=0)
clean_table.head()
```

```
# This will get rid of leadings rows, as well as NaN rows towards the bottom
clean_table = clean_table.iloc[2:41]
clean_table
```

```
# Changing row and column names to appropriate names
clean_table.rename(str.lower, axis='index', inplace=True)
clean_table.rename(str.lower, axis='columns', inplace=True)
# Deleting whitespaces and trailing periods
clean_table.index = clean_table.index.str.replace('.','')
clean_table.index = clean_table.index.str.replace(' ','_')
clean_table.index = clean_table.index.str.lstrip('_')
clean_table.index = clean_table.index.str.rstrip('_')
clean_table.columns = clean_table.columns.str.replace(' ','_')
```

```
clean_table
```

Now, our rows and columns look pretty good. However, we have a couple of different tables combined into one table (cleaned_table). Below, I will attempt to split these into **2 different tables**.

This table shows the **Enrollment in Schools, in Thousands**.

```
# First table
df1 = clean_table.iloc[:19]
df1.style.set_caption("Enrollment in Schools, in Thousands")
df1.index.name = None
df1
```

This table shows the **Percent Distribution in Enrollment in Schools**.

```
# Second table
df2 = clean_table.iloc[20:]
df2.index.name = None
df2
```

Submission 1 - Chapter 2

Process Level 3

This blog post is about the podcast **Scientific Reasoning for Practical Data Science with Andrew Gelmen** to earn **process level 3**. The link to the podcast, if you would like to list, is here.

Today I will be discussing the **Philospohy of Data Science Podcast**. This episode focuses on the practical uses of data science and scientific reasoning, with the overall goal of making key ideas of these topics easier to understand. This episode also has a special guest, **Andrew Gelmen**, who is a Professor of Statistics and Political Science at Columbia University in New York. He has worked on numeruous amounts of projects in applied data science as well as other applications in statistics. Recently, he has several books for linear regression and classification as well

In fact, Andrew Gelmen actually ran his own blog with another researcher in order for them to accurately convey their ideas and findings of their research to one another. This blog also featured a *wiki* section in which other people could get involved and make comments or their own findings on the research. Sadly, the wiki portion of the blog got compromised so they shut down the wiki section altogether. However, the original blog is still live and is located at the following link.

The podcast then goes on to give the listeners/viewers an idea of different classification models, but in simpilier terms. They talk about different modeling tools/classifers such as bayesian modeling, deductive falsification, and so on.

However, a key takeaway from this episode is **why should data scientists care about the philosophy of science?** An important thing about both statistics and data science is that there is no *single way* to do things. The podcast attributes this to the likes of different artisits painting the same model, or different musicians playing the same piece. Although the core concepts are the same, many people have different ways of interpreting data, or what to model or not model.

There are many different kinds of **statistical ideologies**, as the podcast goes into. These kinds of ideologies implant different ideas into people's heads. One such idea could be that *if someone has a randomized controlled experiment with statistical significance, one person could view it differently than the other*. Such as, someone very keen on null hypothesis testing who overvalues results from noisy experiements would view it differently than someone who justifies overally noisy results with outliers.

When considering youir own statistical ideology, there are multiple things to consider:

- You must understand the assumptions you want to make about the data, and why you are making those assumptions
- You must be able to make sense of your own philosophy
- You must be able to connect all of your findings to your larger goals
- You must be able to connect these statements to the world at large to avoid making mistakes by having a narrow mindset or tunnel vision

A lot of these points will help statisticians and data scientists to absolve bias from their data (for the most part) and to also see how their own philosophies and ideologies shape how they interpret their data. Doing this will allow data scientists of all different applied fields (statistics, machine learning, artficial intelligence) to have a greater understanding of data anomalies, and their own **statistical reasoning**.

Submission 1 - Chapter 3

In this submission, I will be attempting to earn **summarize level 3**, **construct level 3**, and **visualize level 3**.

Back in this chapter, I briefly explored the simple folk database. In this Chapter, I plan on diving deeper and creating more tables, as well as visualizing and summarizing data by those created tables.

Summarize Level 3

```
#Imports and data loading
import sqlite3
import pandas as pd
con = sqlite3.connect('simplefolks.sqlite')
cur = con.cursor()
rows = cur.fetchall()
```

This database contains 5 different tables of a simple town. It has information on the town's **people, homes, their pets, politicians, and prison inmates**. Back in Chapter 1, I constructed a couple of tables to show the people and pets of this town, as well as constructing other tables based on simple calculations. In this chapter, I will construct more tables as well as creating new tables based on calculations.

First, lets start with something simple. In this first table, I will find information about the females of this town, and statistics about their age.

```
# First query
query_1 = pd.read_sql_query("SELECT * FROM people WHERE sex = 'F'", con)
query_1
```

```
# Calculating summary statistics of first query
query_1.describe()
```

Now, the above calculation is quite simple. It just tells us the various summary statistics about ALL women in the town, with no other factors. Now, I will create a table with different attributes to create a subset of this first table.

```
# Creating subset of data from first query
subset_1 = pd.read_sql_query("SELECT * FROM people WHERE sex = 'F' AND age >= 35 ORDER BY age DESC",
con)
subset_1
```

The above table describes women in the town whose age is above 35 (close to the mean of the data), with their age in decreases order. This once again is a simple table, but next I will dive into some more complex calculations.

```
# Second query
query_2 = pd.read_sql_query("SELECT * FROM homes", con)
query_2
```

```
# Calculating summary statistics for above data
query_2.describe()
```

This table shows us the homes in the town, who they are owned by, the area they are located in, and the value of the home. The next subset will show us the **three cheapest** homes in all of the different areas.

```
# Subset for urban houses
subset_urban = pd.read_sql_query("SELECT * FROM homes WHERE area = 'urban' ORDER BY value ASC LIMIT
3", con)
# Subset for country houses
subset_country = pd.read_sql_query("SELECT * FROM homes WHERE area = 'country' ORDER BY value ASC
LIMIT 3", con)
# Subset for suburb houses
subset_suburbs = pd.read_sql_query("SELECT * FROM homes WHERE area = 'suburbs' ORDER BY value ASC
LIMIT 3", con)
```

```
# Calculating summary statistics for urban subset
subset_urban.describe()
```

```
# Calculating summary statistics for country subset
subset_country.describe()
```

```
# Calculating summary statistics for suburb subset
subset_suburbs.describe()
```

Based on the summary statistics above, we can see that among the 3 least expensive homes, the minimum value from all the different areas would be in the **country subset** at 42,000. The maximum value from the three least expensive homes at all of the areas would be from the **urban subset** at 190,000. From this data, we can see that the most expensive homes (even from listing the three *least expensive* homes) would be from the **suburb subset** with a minimum of 95,000, and std of 37,527, and a maximum of 160,000.

Now that we are getting more specific, lets list the **3 most expensive homes** that are **not in an urban area** and **not owned by Donald**.

```
# Third subset
subset_3 = pd.read_sql_query("SELECT * FROM homes WHERE area != 'urban' AND owner_name != 'Donald'
ORDER BY value ASC", con)
subset_3
```

Visualize Level 3

Now, let's take the housing data, and create different visuals and plots based on the price, location, and owner.

```
# Imports
import seaborn as sns
import matplotlib.pyplot as plt
```

Showing how many houses are owned by each person.

```
chart_1 = sns.catplot(x="owner_name", kind="count", palette="ch:.25", data=query_2)
chart_1.set_xticklabels(rotation=45)
```

Showing the different values of homes in different areas, note the outliers.

```
chart_2 = sns.catplot(x="value", y="area", kind="box", palette="ch:.25", data=query_2)
```

Showing different homes owned by people and their corresponding values and areas.

```
chart_3 = sns.catplot(x="value", y="owner_name", col="area",
                data=query_2, kind="swarm",
                height=4, aspect=.7, palette="ch:.25");
```

Construct Level 3

In this section I will construct some data tables using data that is not automatically aligned.

```
# Loading data
friends= pd.read_excel("friends.xlsx")
friends_1 = pd.read_excel("friends_1.xlsx")
```

```
# Look at first file
friends
```

```
# Look at second file
friends_1
```

As we can see above, the two datasets are not the same. The first dataset has 7 rows and 5 columns, whereas the second dataset has 5 rows and 4 columns.

The key here is to look for similar column names to merge on, even if the data is not aligned. I will do this below.

```
# First merge
merge_1 = pd.merge(left = friends, right = friends_1, how='outer', on=['name', 'age', 'sport',
'birthday_month'])
merge_1
```

As we can see, we have some missing NaN values in the additional column that was included in the first dataset **friends**. However, we can fill these missing values using the **mode**, or most common occurence of the other values for **favorite_color**.

```
friends.favorite_color.mode()
```

As we can see from the code above, the mode for **favorite_color** in the friends dataset is Orange. Now, we can fill the missing values with these occurences.

```
for column in ['favorite_color']:
    merge_1[column].fillna(merge_1[column].mode()[0], inplace=True)
```

```
merge_1
```

Submission 1 - Chapter 4

For this section of my portfolio, I will be attempting to earn **summarize level 3, visualize level 3, and prepare level 2**. In this first section, I will be correcting **Assignment 4** as well as adding additional graphs and plots to summarize and visualize the data.

Prepare Level 2

For Assignment 4, I was a bit wary of cleaning and preparing data as it was a new concept for me. As I showed briefly in *submission_1*, I have gained some knowledge when dealing with cleaning and preparing data. Similarly, I will be using data from the same database as Assignment 4, located here.

This table shows the different percentages of college students (16-24 years old) who were employed, how many hours they worked a week, and their level of institution.

```
# Imports
import pandas as pd
```

```
# First, let's take a look at what we are dealing with
enroll_uc_table = pd.read_excel("tabn503.20.xls")
enroll_uc_table.head(10)
```

```
enroll_uc_table.tail(10)
```

As we can see from just the table above, the data is quite messy. There are multiply NaN values, headers in the wrong place, as well as trailing characters. Through this submission, I will correct these errors.

```
enroll_clean = pd.read_excel('tabn503.20.xls', skipfooter=7, header=list(range(3)), skiprows=2)
enroll_clean.set_index('Control and level of institution and year',inplace=True)
# enroll_clean = enroll_clean.iloc[2:,]
enroll_clean.dropna('index',inplace=True)
```

```
enroll_clean.head(5)
```

```
# Map years by removing trailing periods
year_mapper = {yr: yr[0].replace('.','').strip() for yr in enroll_clean.index}
year_mapper
```

```
enroll_clean.rename(year_mapper, inplace=True)
enroll_clean.head(10)
```

In this table, we also have to take into account the different levels of students. In this table, we have **4 levels of institutions**. The first is the **total**, the second is **public 4-year institutions**, **private 4-year institutions**, and finally **public 2-year institutions**.

To deal with all of these, I will separate these four different levels into four different tables, as well as having different tables for full-time and part-time students.

```
# Total, all institutions table
total_df = enroll_clean.iloc[:32]
```

```
# Public 4-year institutions table
public_four_df = enroll_clean.iloc[32:43]
```

```
# Private 4-year institutions
priv_4_df = enroll_clean.iloc[43:49]
```

```
# Public 2-year institutions
public_2_df = enroll_clean.iloc[49:]
```

Now, I'm going to split the **total** table into two different dataframes. The first one will be for **full-time students**, while the second one will be for **part-time students**. This will make the overall table much cleaner.

```
# Making full-time dataframe
full_cols = [col for col in total_df.columns if ('Unnamed' in col[2]) or ('Part-time students' in
col[0])
            or ('Less than 20 hours .1' in col[2]) or ('Less than 20 hours .2' in col[2])
            or ('20 to 34 hours .1' in col[2]) or ('20 to 34 hours .2' in col[2])
            or ('35 or more hours .1' in col[2]) or ('35 or more hours .2' in col[2])]
full_time = total_df.drop(full_cols, axis=1)
```

```
# Making part-time dataframe
part_cols = [col for col in total_df.columns if ('Unnamed' in col[2]) or ('Full-time students' in
col[0])
            or ('Less than   20 hours .1' in col[2]) or ('Less than   20 hours .2' in col[2])
            or ('20 to 34 hours .1' in col[2]) or ('20 to 34 hours .2' in col[2])
            or ('35 or more hours .1' in col[2]) or ('35 or more hours .2' in col[2])]
part_time = total_df.drop(part_cols, axis=1)
```

```
# Viewing dataframes
full_time.head()
```

```
part_time.head()
```

Now, we will apply more formatting to the above dataframes to make them cleaner and more readable, also following column name guidelines.

```
# Full-time students dataframe
full_time = full_time.unstack().reset_index().drop('level_1',axis=1)
full_time.rename(columns = {'level_0':'enrollment_status',
                            'level_2':'hours_worked',
                            'Control and level of institution and year':'year',
                           0:'total'}, inplace=True)
full_time.head()
```

```
# Part-time students dataframe
part_time = part_time.unstack().reset_index().drop('level_1', axis=1)
part_time.rename(columns = {'level_0':'enrollment_status',
                            'level_2':'hours_worked',
                            'Control and level of institution and year':'year',
                           0:'total'}, inplace=True)
part_time.head()
```

Now the above tables are much cleaner. To visualize them better, I will develop some plots that will visualize the hours worked in total across full-time and part-time students.

```
# First, concat the two tables together to form the total table
enroll_df = pd.concat([full_time, part_time])
enroll_df
```

Visualize Level 3

```
# Plotting full-time students from total amount
import seaborn as sns
full_g = sns.relplot(data=full_time, x='year', y='total', col='enrollment_status',
hue='hours_worked')
full_g.set_xticklabels(rotation=50)
```

```
# Plotting part-time students from total amount
part_g = sns.relplot(data=part_time, x='year', y='total', col='enrollment_status',
hue='hours_worked')
part_g.set_xticklabels(rotation=50)
```

Summarize Level 3

Now, I'll compute some brief summary statistics regarding the data used in this submission.

```
# Calculating brief summary statistics for full-time students, a subset of the total amount of students
full_time.describe()
```

```
# Calculating various statistics for part-time students, a subset from the total amount of students
part_time.agg({'total' : ['sum', 'min', 'max']})
```

The above summary statistic shows the **sum of the total hours** worked for part-time students, the **minimum number of hours** worked by part-time students, and the **maximum number of hours** worked by a part-time student.

Portfolio Check 2

I've demonstrated skill regression in chapter 1. The file can be found here.
I've demonstrated skill classification, clustering, and evaluate in chapter 2. The file can be found here.
I've demonstrated skill process in chapter 3. The file can be found here.
Please look at Chapter 2 in detail, as I believe I've qualified for both level 2 and 3 for evaluate.

Submission 2 - Chapter 1

I've demonstrated level 3 for regression in this chapter.

Regression Level 3
Can fit and explain nonlinear regression

Regression Level 3

What is non-linear regression?
In the most simple terms, non-linear regression is regression that is **not linear**. However, this also means that non-linear regresssion can be fit to a bunch of different curves because its definition is so fluid. Below I will be fitting a model with both linear and non-linear regresssion.

```
# Imports
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn import datasets
from sklearn.cluster import KMeans
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import PolynomialFeatures
```

```
# Loading data
housing = pd.read_csv("housing.csv")
housing.head()
```

```
# Exploring more about the data
housing.shape
```

```
housing.describe()
```

```
# Looking at correlation between variables
cor = housing.corr()
sns.heatmap(cor)
```

For linear regression analysis, I will be taking a look at the most correlated variable for our *target variable*, which in this case is **median_house_value**.

The msot correlated variable with our target variable seems to be **median_income**.

```
# Adjusting our data
housing = housing.drop(["households","total_bedrooms", "housing_median_age",
"longitude","latitude","total_rooms","population","ocean_proximity"], axis=1)
housing.head()
```

```
# Splitting our data
X = housing.drop("median_house_value", axis=1)
y = housing["median_house_value"]
```

```
X
```

```
y
```

```
plt.scatter(X, y, cmap = "tab20b", alpha=0.25)
```

By looking at the above scatterplot, we can infer that there is a **positive linear relationship** with someone with a hgih income and a more expensive house.

```
# Train, test, split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 0)
```

```
lr = LinearRegression()
lr.fit(X_train, y_train)
```

```
# Predicting values
y_pred = lr.predict(X_test)
y_pred
```

```
# Evaluating model
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2 = r2_score(y_test, y_pred)
```

```
r2
```

```
rmse
```

```
lr.coef_
```

From this we can see that:

- Our R2 score falls in the range between 0 and 1
- Our RMSE could be lower, but it is good enough for regression purposes
- Our model has a slope (coeff) of 42291.42

```
# Visualization of our model
plt.scatter(X_train, y_train, s=10, alpha=0.25)
plt.xlabel('Median Income')
plt.ylabel('Median House Value')
# Plotting predictions
plt.plot(X_test, y_pred, color='g')
plt.show()
```

We can see if the data has any non-linear characteristics by visualizing the data with a **residual plot**. According to seaborn documentation, the **residplot()** function can be a useful tool for checking whether the simple regression model is appropriate for a dataset.

```
# Identifying if the data is non-linear
sns.set_theme(style="whitegrid")
residual = y_test - y_pred
sns.residplot(x=residual, y=y_pred, scatter_kws={'alpha': 0.5}, lowess=True, line_kws={'color':
'green', 'lw': 1, 'alpha': 0.8})
plt.show()
```

The residual line (green) is clearly not straight, which means that a linear model may not be the best fit for our data. In this case, I will try to fit a **polynomial regression model** for our data. Polynomial Regression is defined as a form of regression analysis in which the relationship between the independent variable **x** and the dependent variable **y** are modelled as an nth degree polynomial in **x**.

```
# Fitting the model
poly_reg = PolynomialFeatures()
X_poly = poly_reg.fit_transform(X_train)
pol_reg = LinearRegression()
pol_reg.fit(X_poly, y_train)
```

```
plt.scatter(X_train, y_train, color="green")
plt.plot(X_train, pol_reg.predict(poly_reg.fit_transform(X_train)))
plt.xlabel('Median Income')
plt.ylabel('Median House Value')
plt.show()
```

```
# Predicting values
X_poly = poly_reg.fit_transform(X_test)
y_pred_2 = pol_reg.predict(X_poly)
```

```
# Evaluating the model
rmse = np.sqrt(mean_squared_error(y_test, y_pred_2))
r2 = r2_score(y_test, y_pred_2)
```

```
r2
```

```
rmse
```

By using this new model, our R2 score increased from **0.4564966485656323** with the Linear Regression model, to **0.4628233152699195** with the Polynomial Regression model.

Submission 2 - Chapter 2

I've demonstrated **classification level 3**, **evaluate level 2 (and 3, if applicable)** and **clustering level 3** in this chapter.

Classification
Fit and apply classification models and select appropriate classification models for different contexts

Clustering
Apply multiple clustering techniques, and interpret results

Evaluate
Level 2: Apply basic model evaluation metrics to a held out test set
Level 3: Evaluate a model with multiple metrics and cross validation

In this chapter I will be looking at the same dataset I explored in *Assignment #9* previously. I'll include some of the same EDA and methods as well.

Classification Level 3

```
# Imports
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.cluster import KMeans
import pandas as pd
from sklearn.metrics import classification_report,accuracy_score, silhouette_score,
adjusted_rand_score, adjusted_mutual_info_score
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
```

```
# Loading data
wine_data = pd.read_csv("winequality.csv")
wine_data.head()
```

```
wine_data.shape
```

```
# Visualizations of data before
plt.figure(figsize=(10,10))
sns.heatmap(wine_data.corr(),annot=True,linewidth=0.5,center=0,cmap='flare')
plt.show()
```

```
# More visualization
sns.pairplot(wine_data, vars=['chlorides','density', 'alcohol'], hue='quality', palette="flare")
```

```
# Exploring classifications: by quality
print(wine_data['quality'].unique())
```

```
# More visualization of classes with qualtiy and alcohol
sns.FacetGrid(wine_data, hue="quality", height=6, palette="flare").map(plt.scatter, "alcohol",
"density").add_legend()

plt.show()
```

```
# Encoding color as 0=white, 1=red
wine_data.color.apply(lambda x: 1 if x == "red" else 0)
wine_data
```

```
# Separating Quality into High/Low
quality = wine_data["quality"].values
temp = []
for num in quality:
    if num<5:
        temp.append("Low")
    else:
        temp.append("High")
temp = pd.DataFrame(data=temp, columns=["ranking"])
data = pd.concat([wine_data,temp],axis=1)
data.drop(columns="quality",axis=1,inplace=True)
```

```
data.head()
```

First Method: KNN Classification

```
# Separating target from feature variables
X= data.iloc[:,:-1].values
y=data.iloc[:,-1].values
```

```
# Train, test, split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,random_state=0)
```

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
```

```
pred_knn=knn.predict(X_test)
print(classification_report(y_test, pred_knn))
```

Wow! The accuracy is really high with this classifier. Lets explore other options just in case.

Second Method: Random Forest

```
# Train, test, split
X_train2, X_test2, y_train2, y_test2 = train_test_split(X, y, test_size = 0.2,random_state=0)
```

```
# Building the forest
rfc = RandomForestClassifier()
rfc.fit(X_train2, y_train2)
pred_2 = rfc.predict(X_test2)
print(classification_report(y_test2, pred_2))
```

Third Method: Naive Bayes

```
# Train, test, split
X_train3, X_test3, y_train3, y_test3 = train_test_split(X, y, test_size = 0.2,random_state=0)
```

```
# Applying NB
gnb = GaussianNB()
gnb.fit(X_train3, y_train3)
y_pred3 = gnb.predict(X_test3)
```

```
gnb.score(X_test3,y_test3)
```

```
gnb.predict_proba(X_test3)
```

```
print(classification_report(y_test3, y_pred3))
```

```
# Combining Results of Classifiers
results = pd.DataFrame({'model': ["KNN","Random Forest","Naive Bayes"],
                        'accuracies':
[accuracy_score(y_test,pred_knn),accuracy_score(y_test2,pred_2),accuracy_score(y_test3,y_pred3)]})
results
```

Based on the table above, the best classifier for the **Wine Quality** Dataset would be the **Random Forest** classifier, which has a very tiny difference between the **KNN** classifier.

Clustering Level 3

Method 1: Using KMeans

I previously used KMeans for clustering in *Assignment #9*, so I'll briefly touch upon that now and then explore other methods.

```
# Separating target from feature variables
X= data.drop(["ranking"], axis = 1)
y=data["ranking"]
```

```
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
cent = kmeans.cluster_centers_
```

```
y_kmeans = kmeans.fit_predict(X)
y_kmeans
```

```
centroids_df = pd.DataFrame(kmeans.cluster_centers_, columns = list(X.columns.values))
```

```
# Plotting relationship between alcohol and density when predicting quality of wine
fig, ax = plt.subplots(1, 1)
X.plot.scatter(x='alcohol', y='density', c= y_kmeans, figsize=(10,8), colormap='flare', ax=ax,
mark_right=False)
centroids_df.plot.scatter(x='alcohol', y='density', ax = ax, s = 80, mark_right=False)
```

```
# Compiling different metrics
metrics1 = pd.DataFrame({'metric': ["Silhouette","Rand Score","Mutual Info"],
                        'score': [silhouette_score(X, y_kmeans),adjusted_rand_score(y,
y_kmeans),adjusted_mutual_info_score(y,y_kmeans)]})
metrics1
```

Method 2: Using Mean Shift

```
from sklearn.cluster import MeanShift, estimate_bandwidth
bandwidth = estimate_bandwidth(X)
msc = cluster.MeanShift(bandwidth=bandwidth, bin_seeding=True)
```

```
msc.fit(X)
```

```
labels = msc.labels_
cluster_centers = msc.cluster_centers_
```

```
labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)
print("number of estimated clusters : %d" % n_clusters_)
```

```
y_msc = msc.predict(X)
y_msc
```

```
centroids_df = pd.DataFrame(cluster_centers, columns = list(X.columns.values))
centroids_df.head()
```

```
# Plotting relationship between alcohol and density when predicting quality of wine
fig, ax = plt.subplots(1, 1)
X.plot.scatter(x='alcohol', y='density', c= y_msc, figsize=(10,8), colormap='flare', ax=ax,
mark_right=False)
centroids_df.plot.scatter(x='alcohol', y='density', ax = ax, s = 80, mark_right=False)
```

```
# Compiling different metrics
metrics2 = pd.DataFrame({'metric': ["Silhouette","Rand Score","Mutual Info"],
                         'score': [silhouette_score(X, y_msc),adjusted_rand_score(y,
y_msc),adjusted_mutual_info_score(y,y_msc)]})
metrics2
```

Method 3: Agglomerative Clustering

```
from sklearn.cluster import AgglomerativeClustering
ac = AgglomerativeClustering(n_clusters=5)
ac.fit(X)
```

```
ac.labels_
```

```
y_ac = ac.fit_predict(X)
```

```
# Plotting relationship between alcohol and density when predicting quality of wine
fig, ax = plt.subplots(1, 1)
X.plot.scatter(x='alcohol', y='density', c= y_ac, figsize=(10,8), colormap='flare', ax=ax,
mark_right=False)
```

```
# Compiling different metrics
metrics3 = pd.DataFrame({'metric': ["Silhouette","Rand Score","Mutual Info"],
                         'score': [silhouette_score(X, y_ac),adjusted_rand_score(y,
y_ac),adjusted_mutual_info_score(y,y_ac)]})
metrics3
```

Now, I will compile all of the different accuracies, along with the scoring method used for all of the different kinds of clustering.

```
# Combining all dataframes
result1 = pd.merge(left = metrics1, right = metrics2, how='inner', on=["metric"])
result = pd.merge(left = result1, right = metrics3, how='inner', on=["metric"])
result.rename(columns={"score_x":"K-means", "score_y":"Mean Shift", "score":"Agglomerative
Clustering"})
```

From the above table, we can see that **K-Means Clustering** had an overall better silhouette score, **Mean Shift** had a better **Mutual Info** score, and **Agg. Clustering** had a better **Rand Score**.

Evaluate Level 2/3

You can see from the multiple metrics I used throughout this portion (both in the clustering and classification sections) that this warrants **evaluate level 2** for applying basic model evaluation metrics, and *part* of **level 3** for using multiple metrics with multiple different types of classifiers.

Evaluate Level 3

I will now perform some cross-validation for the **Iris Dataset** shown in class, along with using different train and test sizes for the data.

```
# Loading data
iris_df = pd.read_csv('Iris.csv')
```

```
# Brief visualization
sns.pairplot(iris_df, hue='Species', palette="flare")
```

```
# Target and feature variables
X = iris_df[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']].values
y = iris_df['Species'].values
```

```
# Using a loop to change test sizes, correction from previous assignments!
from sklearn.metrics import accuracy_score
testsize_range = [n/10 for n in range(1,7)]
testsize_scores = []
dt = DecisionTreeClassifier()
for testsize in testsize_range:
    X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = testsize, random_state=0)
    dt.fit(X_train, y_train)
    y_pred = dt.predict(X_test)
    scores = (accuracy_score(y_test, y_pred))
    testsize_scores.append(scores.mean())
```

```
final_results = pd.DataFrame({'test_size': ["0.1","0.2","0.3", "0.4", "0.5", "0.6"],
                              'score': testsize_scores})
final_results
```

As we can see, it seems that a **test_size of 0.3 seems to work best for the Iris dataset**. Accurracys of 1 might mean that the dataset is too overfit, or it might be overtrained.

Submission 2 - Chapter 3

I've demonstrated skill *process* in this chapter.

Process
Compare different ways that data science can facilitate decision making

Process Level 3

There are many different ways that data science can facilitate decision making. For example, individuals in the field of economics now have access to more data about consumer insights and consumption patterns. These newfound analytics can be used to shape their products, solutions, customer service, and overall buying experience. According to this research study, organizations that are using their consumer behavior insights strategically are outperforming their peers by 85 percent in sales growth margins and by more than 25 percent in gross margins.

Other than applications in the business and economic sectors, data science can also facilitate decision making when it comes to identifying diseases and disorders using machine learning. From the models we learned in class (and different applications I've used in assignments), ML can be used to aid the process of *detecting benign vs. cancerous tumors*, *identifying*

asymptomatic vs. symptomatic patients, and even giving the entire diagnose based on a set of chosen attributes and features.

The process of decision making has been greatly aided by the introduction and incorporation of Data Science practices and methods. From implementations of machine learning, statistics, and domain expertise, all different sectors of Data Science help to facilitate decision making in a variety of fields.

Portfolio Check 4

I've demonstrated skills workflow level 3, clustering level 3, evaluate level 3, compare level 2 and 3, and optimize level 3 in chapter 1. The file can be found here.
I've demonstrated skill unstructured level 3. The file can be found here.

Please look at Chapter 1 in detail, as many of the sections have areas of other skills in them (which I do specify).

Thank you for the great semester!

Submission 4 - Chapter 1

I've demonstrated **workflow level 3**, **clustering level 3**, **evaluate level 3**, **compare level 2 and 3**, and **optimize level 3** in this chapter.

Workflow Level 3
Scope, choose an appropriate tool pipeline and solve data science problems, describe strengths and weakensses of common tools.

Clustering Level 3
Apply multiple clustering techniques, and interpret results

Evaluate Level 3
Evaluate a model with multiple metrics and cross validation.

This skill is present in the multiple metrics used to assess the different types of clustering methods, as well as performing cross-validation in the Iris dataset.

Compare Level 2
Compare model classes in specific terms and fit models in terms of traditional model performance metrics.

Compare Level 3
Evaluate tradeoffs between different model comparison types.

Optimize Level 3
Select optimal parameters based of mutiple quantitative criteria and automate parameter tuning.

Workflow Level 3

For this section, I will go over my process for the next section (which involves clustering and other skills). I will also go over what I want to achieve using these models, and other tools that could also be used to replace what I am using.

Common Tools in Data Science
There are many different tools that can be used and applied in data science (whether for machine learning purposes, or just statistical). I will list some of these tools that I have personally used before below:

- SKLearn
- SciPy
- Matplotlib
- Seaborn

Why did I choose the tools I did for this assignment?

SKLearn vs. SciPy
For this assignment I chose to use a vast majority of methods/classifers from **SKLearn**. In this class, I primarily use SKlearn,

but I have used Scipy before. SKLearn is definitely easier, as its documentation provides fully thoguht-out examples and very clear source code of their functions/methods. Scipy is usually more devoted to deeper kinds of calculations (I've used it previously for fourier transformations and signal processing). However, because this class primarily uses SKLearn, I chose to also use that for the majority of my portfolio.

Strengths of SKLearn:

- Main strengths are accessiblity, adaptability, and simplicity.
- Very easy for beginners to grasp basic concepts.
- License allows you to upstream changes vwithout restrictions on commerical use.

Strengths of SciPy

- Open-source, does not cost anything
- Available libraries make it easy to convert to C or Fortran code
- Has a lot of correlating libraries with Numpy which makes it easy to use the two together

Seaborn vs. Matplotlib
For this assignment, I used both matplotlib and seaborn. I usually like to add more customizations using matplotlib (such as titles, axis rotation, etc) but I prefer the different colore palettes of seaborn. When choosing a tool, you have to keep in mind what is best for visualizing your data. Both seaborn and matplotlib provide an excellent amount of different visualizations, but each different library can provide better visualizations than others. For example, I prefer using **matplotlib's scatterplot function** over seaborn's because of the usability and aesthetics.

Strengths of Seaborn

- Use default themes that are aesthetically pleasing.
- You can set custom color palettes.
- Makes attractive statistical plots.
- Easily and flexible for displaying distributions.

Strengths of Matplotlib

- Can be automated to adapt to the data that it receives as input.
- Simple and easy to grasp for beginners.
- asier to use for people who have had prior experience with Matlab.

Now, I will talk about my process for the **Clustering** section of this portfolio (which is the next section).

For this section, I wanted to predict the quality of wine (high/low) depending on different predictors. To do this, I had to:

- Encode the color of the wine to make it easier to manipulate, along with other categorical variables.
- Define what a 'high' anmd 'low' quality wine was, without being too biased.
- Determine what type of clustering I wanted to use, and how it performed.
- Determine if there was a better way to come to this prediction (in terms of other models, or different types of algorithms).

I enjoy making a list of what I want to accomplish before I start working on a problem, so this is why this section comes first in this portfolio.

I explain more about why I chose my clustering methods in the **Compare Section** of this portfolio, which I believe should also count for **Workflow :)**.

Clustering Level 3

I was only missing the descriptions of the clustering techniques to earn this skill in my last check, so I will add descriptions as necessary.

```
# Imports
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.cluster import KMeans
import pandas as pd
from sklearn.metrics import classification_report,accuracy_score, silhouette_score,
adjusted_rand_score, adjusted_mutual_info_score
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import average_precision_score
```

```
# Loading data
wine_data = pd.read_csv("winequality.csv")
wine_data.head()
```

```
wine_data.shape
```

```
# Exploring classifications: by quality
print(wine_data['quality'].unique())
```

```
# Encoding color as 0=white, 1=red
wine_data["color"] = wine_data.color.apply(lambda x: 1 if x == "red" else 0)
wine_data
```

```
# Separating Quality into High/Low
quality = wine_data["quality"].values
temp = []
for num in quality:
    if num<5:
        temp.append("Low")
    else:
        temp.append("High")
temp = pd.DataFrame(data=temp, columns=["ranking"])
data = pd.concat([wine_data,temp],axis=1)
data.drop(columns="quality",axis=1,inplace=True)
```

```
data.head()
```

Method 1: Using KMeans

What is K-Means Clustering?

- Partitions datasets into K defined subgroups (non-overlapping) called clusters
- Each data point can only belong to one group
- K-Means then assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid is at the minimum

How does K-Means Clustering Work?

- Specify the number of clusters (K)
- Intialize centroids by selecting K points for the centroids
- Iterate until there is no change in the centroids, i.e. the clusters are not changing anymore

Source:

- https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a

```
# Separating target from feature variables
X= data.drop(["ranking"], axis = 1)
y=data["ranking"]
```

```
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
cent = kmeans.cluster_centers_
```

```
y_kmeans = kmeans.fit_predict(X)
y_kmeans
```

```
centroids_df = pd.DataFrame(kmeans.cluster_centers_, columns = list(X.columns.values))
```

```
# Plotting relationship between alcohol and density when predicting quality of wine
fig, ax = plt.subplots(1, 1)
X.plot.scatter(x='alcohol', y='density', c= y_kmeans, figsize=(10,8), colormap='flare', ax=ax,
mark_right=False)
centroids_df.plot.scatter(x='alcohol', y='density', ax = ax, s = 80, mark_right=False)
```

```
# Compiling different metrics
metrics1 = pd.DataFrame({'metric': ["Silhouette","Rand Score","Mutual Info"],
                         'score': [silhouette_score(X, y_kmeans),adjusted_rand_score(y,
y_kmeans),adjusted_mutual_info_score(y,y_kmeans)]})
metrics1
```

Method 2: Using Mean Shift

What is Mean Shift Clustering?

- Mean Shift is formed on the idea of KDE (kernel density estimation) which is a method used to estimate the underlying distribution of the data.
- A kernel/weighting function is placed on each point in the dataset, and then updates the centroids of the dataset based on the processing of the data.
- Aims to discover “blobs” in a smooth density of samples. It is a centroid-based algorithm, which works by updating candidates for centroids to be the mean of the points within a given region.

How does Mean Shift Clustering Work?

- At every iteration the kernel is shifted to the centroid or the mean of the points within it.
- The method of calculating this mean depends on the choice of the kernel. In this case if a Gaussian kernel is chosen instead of a flat kernel, then every point will first be assigned a weight which will decay exponentially as the distance from the kernel’s center increases.
- At convergence, there will be no direction at which a shift can accommodate more points inside the kernel, and iteration most likely will end.

Sources:

- https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MeanShift.html
- https://www.sciencedirect.com/science/article/pii/S0047259X14002644?via%3Dihub

```
from sklearn.cluster import MeanShift, estimate_bandwidth
bandwidth = estimate_bandwidth(X)
msc = MeanShift(bandwidth=bandwidth, bin_seeding=True)
```

```
msc.fit(X)
```

```
labels = msc.labels_
cluster_centers = msc.cluster_centers_
```

```
labels_unique = np.unique(labels)
n_clusters_ = len(labels_unique)
print("number of estimated clusters : %d" % n_clusters_)
```

```
y_msc = msc.predict(X)
y_msc
```

```
centroids_df = pd.DataFrame(cluster_centers, columns = list(X.columns.values))
centroids_df.head()
```

```
# Plotting relationship between alcohol and density when predicting quality of wine
fig, ax = plt.subplots(1, 1)
X.plot.scatter(x='alcohol', y='density', c= y_msc, figsize=(10,8), colormap='flare', ax=ax,
mark_right=False)
centroids_df.plot.scatter(x='alcohol', y='density', ax = ax, s = 80, mark_right=False)
```

```
# Compiling different metrics
metrics2 = pd.DataFrame({'metric': ["Silhouette","Rand Score","Mutual Info"],
                         'score': [silhouette_score(X, y_msc),adjusted_rand_score(y,
y_msc),adjusted_mutual_info_score(y,y_msc)]})
metrics2
```

Method 3: Agglomerative Clustering

What is Agglomerative Clustering?

- Partitions datasets into K defined subgroups (non-overlapping) called clusters
- One of the most common types of hierarchial clustering
- Agglomerative clustering works in a "bottom-up" manner. (objects are considered leafs initially, and then are combined into nodes (clusters).

How does Agglomerative Clustering Work?

- Starts by treating each object as a singleton cluster.
- Pairs of clusters are successively merged until all clusters have been merged into one big cluster containing all objects.
- The result is a tree-based representation of the objects, named dendrogram.

Sources:

- https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html
- https://www.datanovia.com/en/lessons/agglomerative-hierarchical-clustering/

```
from sklearn.cluster import AgglomerativeClustering
ac = AgglomerativeClustering(n_clusters=5)
ac.fit(X)
```

```
ac.labels_
```

```
y_ac = ac.fit_predict(X)
```

```
# Plotting relationship between alcohol and density when predicting quality of wine
fig, ax = plt.subplots(1, 1)
X.plot.scatter(x='alcohol', y='density', c= y_ac, figsize=(10,8), colormap='flare', ax=ax,
mark_right=False)
```

```
# Compiling different metrics
metrics3 = pd.DataFrame({'metric': ["Silhouette","Rand Score","Mutual Info"],
                         'score': [silhouette_score(X, y_ac),adjusted_rand_score(y,
y_ac),adjusted_mutual_info_score(y,y_ac)]})
metrics3
```

Now, I will compile all of the different accuracies, along with the scoring method used for all of the different kinds of clustering.

```
# Combining all dataframes
result1 = pd.merge(left = metrics1, right = metrics2, how='inner', on=["metric"])
result = pd.merge(left = result1, right = metrics3, how='inner', on=["metric"])
result.rename(columns={"score_x":"K-means", "score_y":"Mean Shift", "score":"Agglomerative
Clustering"})
```

From the above table, we can see that **K-Means Clustering** had an overall better silhouette score, **Mean Shift** had a better **Mutual Info** score, and **Agg. Clustering** had a better **Rand Score**.

Evaluate Level 3

I will now perform some cross-validation for the **Iris Dataset** shown in class, along with using different train and test sizes for the data.

```
# Loading data
iris_df = pd.read_csv('Iris.csv')
```

```
# Brief visualization
sns.pairplot(iris_df, hue='Species', palette="flare")
```

```
# Target and feature variables
X = iris_df[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']].values
y = iris_df['Species'].values
```

Now, let's evaluate using **cross validation~** as well as **accuracy scores**, Which I always forget to do, but I am definitely remembering to do this time.

```
# Using a loop to change test sizes
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
testsize_range = [n/10 for n in range(1,7)]
testsize_scores_dt = []
crossval_scores_dt = []
dt = DecisionTreeClassifier()
for testsize in testsize_range:
    X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = testsize, random_state=0)
    dt.fit(X_train, y_train)
    y_pred = dt.predict(X_test)
    # See, I remembered this time
    cross_val = cross_val_score(dt, X, y)
    scores = (accuracy_score(y_test, y_pred))
    testsize_scores_dt.append(scores.mean())
    crossval_scores_dt.append(cross_val.mean())
```

```
final_results = pd.DataFrame({'test_size': testsize_range,
                              'score': testsize_scores_dt, "cross validation": crossval_scores_dt})
final_results
```

As we can see, it seems that a **test_size of 0.3 seems to work best for the Iris dataset**. Accurracys of 1 might mean that the dataset is too overfit, or it might be overtrained. The **cross validation score** for test_size 0.3 also fits within the accepted range, so this test size would work perfectly.

Just to make sure I absolutely get **Level 3 for Evaluate**, I will perform KNN on this same dataset and then also perform cross-validation 15 times.

```
# Using similar loop from before
from sklearn.neighbors import KNeighborsClassifier
k_range = list(range(1,40,2))
crossval_scores_knn = []
acc_scores_knn = []
# Using 15 fold cross validation
for k in k_range:
    X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.3, random_state=0)
    # See, I remembered this time
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(X_train, y_train)
    # Cross Val score
    cross_val = cross_val_score(knn, X_train, y_train, cv=15)
    crossval_scores_knn.append(cross_val.mean())
    # Accuracy Score
    accuracy_score_knn = knn.score(X_train, y_train)
    acc_scores_knn.append(accuracy_score_knn)
```

```
# Compiling into dataframe
cv_results = pd.DataFrame({'k': k_range, "cross validation": crossval_scores_knn,
                           "accuracy": acc_scores_knn})
cv_results
```

We can see that **K=7** has the best cross validation score **AND** accuracy (CV Score = 0.961905, Acc = 0.971429).

```
from sklearn.neighbors import KNeighborsClassifier
# Using our optimal test size from earlier, as well as our optimal k
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state=0)
knn2 = KNeighborsClassifier(n_neighbors=7)
knn2.fit(X_train, y_train)
# Predicting values
pred_knn2=knn2.predict(X_test)
print(classification_report(y_test, pred_knn2))
```

```
# Computing CV again just to show
cross_val = cross_val_score(knn2, X_train, y_train, cv=15)
score_knn_1 = (accuracy_score(y_test, pred_knn2))
print(cross_val)
```

Compare Level 2

For this section, I will be comparing the two above models (Decision Trees and KNN, as well as Naive Bayes just for fun), as well as the different metrics used to evaluate them, as well as the trade-offs when applying them.

Comparing Decision Trees, Naive Bayes, and KNN in Specific Terms
Both methods are used for classification, but there are many differences in the three models.

Naive Bayes

- Supervised learning.
- Linear classifer (unlike KNN).
- Highly accurate when applied to *big data* or large feature sets.
- More hyperparameters than KNN (alpha and beta).
- Does not suffer from the curse of dimensionality.

Decision Trees

- Supervised learning.
- Easiest to explain and understand.
- Over-fitting is a major problem with decision trees.
- Works best for a small number of classes.
- Outperformed by KNN and GNB when it comes to *rare occurences* such as outliers, as decision trees sometime require pruning.

KNN

- Unsupervised learning.
- All the features must be numeric.
- Doesn't require training.
- You would want to choose KNN over GNB if there is high conditional independence among predictors.
- Suffers from the curse of dimensionality in some cases (more variables/features leads to sample size growing exponentially).

In The past section involving the **Iris Dataset**, I have used both a Decision Tree Classifer, Naive Bayes Classifer, as well as a KNN Classifer to yield results.

Previously for the DT Classifer and the KNN Classifer, I assessed the two model's results using the *score method* as well as the *cross_val_score* function. Below I will compile a dataframe with the different metrics from both of these models, as well as getting new metrics for KNN and GNB.

```
# Comparing accuracy/cross_vcal_score of test size for KNN
# Using a loop to change test sizes (same as bework best for a small number of classesfore, but for
knn)
testsize_range = [n/10 for n in range(1,7)]
testsize_scores_knn = []
crossval_scores_knn = []
knn = KNeighborsClassifier()
for testsize in testsize_range:
    X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = testsize, random_state=0)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    # See, I remembered this time
    cross_val = cross_val_score(knn, X, y)
    scores = (accuracy_score(y_test, y_pred))
    testsize_scores_knn.append(scores.mean())
    crossval_scores_knn.append(cross_val.mean())
```

```
# Comparing accuracy/cross_vcal_score of test size for NB
testsize_range = [n/10 for n in range(1,7)]
testsize_scores_gnb = []
crossval_scores_gnb = []
gnb = GaussianNB()
for testsize in testsize_range:
    X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = testsize, random_state=0)
    gnb.fit(X_train, y_train)
    y_pred = gnb.predict(X_test)
    # See, I remembered this time
    cross_val = cross_val_score(gnb, X, y)
    scores = (accuracy_score(y_test, y_pred))
    testsize_scores_gnb.append(scores.mean())
    crossval_scores_gnb.append(cross_val.mean())
```

```
final_results_knn = pd.DataFrame({'test_size': testsize_range,
                          'score': testsize_scores_knn, "cross validation": crossval_scores_knn})
final_results_gnb = pd.DataFrame({'test_size': testsize_range,
                          'score': testsize_scores_gnb, "cross validation": crossval_scores_gnb})
```

```
results = pd.merge(left = final_results, right = final_results_knn, on = ["test_size"])
results_final = pd.merge(left = results, right = final_results_gnb, on = ["test_size"])
results_final.rename(columns={"score_x":"Score DT", "score_y":"Score KNN",
                        "cross validation_x":"Cross Val Score DT",
                        "cross validation_y":"Cross Val Score KNN",
                        "score":"Score GNB",
                        "cross validation":"Cross Val Score GNB"})
```

As we can see, our Decision Tree did suffer through some *mild overfitting*, as discovered in the previous section. However, it had better **cross_val_score** than the GNB model with the range of test sizes that it was used on. In terms of consistent model performance, KNN performed very well in terms of **cross_val_score** and **score/accuracy**. This is examined more in the previous sections as well. Unlike for other datasets that I've used in the past, the Naive Bayes model did not perform as well as anticipated. This is probably due to the fact that the Iris dataset is not that large, and is perfect in terms of small classes (3 different types of Irises) for the KNN classifer.

Compare Level 3

Now I will evaluate the different tradeoffs that can be made by choosing from the models discussed in the previous *Compare* section (Decision Trees, Naive Bayes, KNN).

Overall, each of these models has a decision to make in terms of the **bias variance trade-ff** This tradeoff is simply explained as:

- The model is too simple is often reflected in a biased model with fewer features and more regularization.
- The model is too complex when small changes are made that affect the data tremendously, due to high variance with more features and less regularization.

So what do you choose?

- More complex with less bias and more variance?
- More simple with more bias and less variance?

Trade-Offs of KNN

- There is a definite **bias-variance trade-off** when it comes to KNN classification. This means that if bias is reduced, the variance is increased and vice versa.

- *Bias* is caused by highly correlated predictors, or misinterpretting the relationships between the features and the targets in a dataset.

- *Variance* is caused by fluctuations in the training set. High variance can lead to the model having random noise in the data, and leading thje model to pay a lot of attention to the training data and in turn offer fewer generalizations on data that it has not yet seen.

- Together, bias and variance can lead to the model becoming **over or under - fitted**. How can we solve this?

Solution for KNN:

- **Cross Validation**, as we have used previously in this Chapter, can introduce a **validation data set** along with the training and testing sets a model usually has.
- Considering what is important in your model can also help in discovering what is more important in terms of bias and variance. Ideally, low bias and low variance would make for a very well-fitted model, but that can not always occur. This is why introducing a validation set is a good solution.

Trade-Offs of Decision Trees
Similarly to KNN, Decision Trees also face a **bias variance trade-off**. However, solving this problem is a little bit different than KNN. **Decision Trees are also known to have *high variance*, as they create specific branches and splits for samples of the training data, that are specific to this data.**

- In a DT, more roughness = more variance while more smoothness = more bias. When a model gets rougher, it gets more complex. When a model gets smoother, it gets less complex.
- To combat this problem, **pruning the tree** is used to compress the data and to reduce the dimensionality of the model, while still keeping complexity.
- Decreasing the accuracy of the model on the training data increases bias. This lowers variance, meaning your model better generalizes to unseen data (as explained in the KNN section).

Trade-Offs of Naive Bayes
Like other models discussed in this section, Naive Bayes also suffers from **bias variance trade-off**. There are also some solutions to this, such as **tuning hyperparameters alpha and beta**, which I will do in the next section for **Level 3 for Optimize**. Here are the different trade-offs for GNB:

- If alpha = 0 (high variance): our model is overfitted
- If alpha is large (high bias): posterior probabilities are the same

Other Solutions: Trying different Variations of NB There are different types of Naive Bayes classifiers that we can choose from, including:

- Bernoulli NB
- Multinomial NB
- Gaussian NB (which we used above)

Finally, general solutions for solving these trade-offs:
With a model with **high bias:**

- Change the optimization algorithm of the model.
- Perform hyper-parameter tuning on the model.
- Switch the type of model.

With a model with **high variance:**

- Perform regularization using pruning (for DT), dropout (KNN/NB), or Lasso/Ridge Regression for Regression Models.
- Get more data to train on
- Try a different model type

Optimize Level 3

Going off of the previous section and the criteria for the KNN model, I will now optimize the parameters of our model to insure we have the lowest variance and the lowest bias possible.

To do this, I will optimize the following values:

- n_neighbors, as before
- weights
- p (power parameter)

I optimized these parameters briefly when comparing models in the earlier sections, but I want to agaijn try to optimize them even further.

I will be using a **GridSearch** to perform the tuning of these hyperparameters.

```
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
# Default is uniform
weights = ["uniform", "distance"]
# Default 2 (euclidean distance)
p = [1, 2]
# Number of neighbors, default 5, as before (same range)
k_range = list(range(1,40,2))

param_grid = {"n_neighbors": k_range,
              "weights": weights,
              "p": p
              }

grid = GridSearchCV(estimator = KNeighborsClassifier(),
                    param_grid = param_grid,
                    cv = 15, # 15, as used previously in the above section
                    scoring = 'accuracy',
                    refit = True)

knn_model = make_pipeline(StandardScaler(), grid)
knn_model.fit(X_train, y_train)

print(grid.best_score_)
print(grid.best_params_)
```

We can see that with this optimization, the nubmer of neighbors for our KNN classifer is optimized at **3**, our p is optimized at **1 (manhattan_distance)**, and our weights are optimized to be **uniform**.

Now, I will fit a model with these optimized paramters and compare it to the un-optimized version.

```
# Comparing optimized vs. un-optimized KNN model
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state=0)
knn3 = KNeighborsClassifier(n_neighbors=3, weights="uniform", p=1)
knn3.fit(X_train, y_train)
# Predicting values
pred_knn3=knn3.predict(X_test)
print(classification_report(y_test, pred_knn3))
```

```
# Cross-val score and accuracy score for optimized model
# Optimized Model
cross_val_opt = cross_val_score(knn3, X_train, y_train, cv=15).mean()
scores_opt = (accuracy_score(y_test, pred_knn3)).mean()
# Un-optimized Model
cross_val = cross_val.mean()
score_knn_1 = score_knn_1.mean()
```

```
# Comparing the two models
compare_results = pd.DataFrame({'Cross_Val_Score': [cross_val, cross_val_opt],
                            'Accuracy': [score_knn_1, scores_opt]})
compare_results
```

We can see that the two models compare very similarly to each other, but this is also because the other model was optmized for test_size while our newly optimized model using GridSearch did not account for test size. I think it would be better to also add that in as well.

In Evaluate Level 3/Compare Level 2 sections, I automated the tuning of hyperparameters using loops. For this section I wanted to use GridSearch to showcase other ways you could tune these hyperparameters. To consider the automation of these parameters, please look at the aformentioned sections I listed here :)

Submission 4 - Chapter 2

In this chapter I have demonstrated **unstructured level 3**.

Unstructured Level 3
Apply multiple representations and compare and contrast them for different end results.

Unstructured Level 3

For this portion of my portfolio, I will be briefly touching upon the different representations of unstructured data covered in **Assignment 12**.

To refresh, here are the different methods we can use:

CountVectorizer
Returns an encoded vector with a length of the entire vocabulary and an integer count for the number of times each word appeared in the document.

TfidfVectorizer
Allows use to tokenize documents, learn similar vocabularies and the frequencies/weights of different works in documents (but not across documents), and allows you to encode new documents.

HashingVectorizer
Uses a one-way hash of words to convert them into integers, and then tokenizes and encodes the documents as needed.

For this portfolio, I will be comparing and contrasting **CountVectorizer** with **TfidVectorizer**. Here are the **similarities and differences** between these two methods:

- TfidVectorizer returns a float instead of an int like CountVectorizer.
- In TfidfVectorizer we consider overall document weightage of a word. It helps us in dealing with most frequent words.
- In CountVectorizer we only count the number of times a word appears in the document which results in biasing in favour of most frequent words.

For this assignment, I will be completing Assignment 12 with both **CountVectorizer** and **TfidVectorizer**, and then comparing the results.

```
# Imports
from sklearn.metrics import accuracy_score
from math import sqrt
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.model_selection import train_test_split
import pandas as pd
from sklearn.metrics.pairwise import euclidean_distances
import numpy as np
```

Using CountVectorizer

```
# Loading Data
news = pd.read_csv("fake_or_real_news.csv")
news.head()
```

```
# Dropping labels that are not necessary
news.drop(labels=['id','title'], axis='columns', inplace=True)
news.head()
# Separate text from labels
text = news['text']
labels = news['label']
# Vectorize Data
counts = CountVectorizer(analyzer = "word")
ng_vec = counts.fit_transform(text).toarray()
# get_feature_names method will return them as a sorted list instead of a dictionary with numbers.
counts_df = pd.DataFrame(ng_vec, columns=counts.get_feature_names())
```

```
# Train, test, split
X_train, X_test, y_train, y_test = train_test_split(counts_df, labels, test_size = 1000)

# Naive Bayes Model
clf = MultinomialNB()
clf.fit(X_train, y_train)
# Scoring the model
clf.score(X_test, y_test)
# Making predictions on test data
y_pred_test = clf.predict(X_test)
# Making predictions on training data
y_pred_train = clf.predict(X_train)
# Training Accuracy
train_acc = accuracy_score(y_train, y_pred_train)
# Testing Accuracy
test_acc = accuracy_score(y_test,y_pred_test)
```

```
train_acc
```

```
test_acc
```

Using TfidVectorizer

```
# Vectorize Data
tfid = TfidfVectorizer(analyzer = "word")
tfid_vec = tfid.fit_transform(text).toarray()
tfid_df = pd.DataFrame(tfid_vec, columns=tfid.get_feature_names())
```

```
# Train, test, split for TFID
X_train, X_test, y_train, y_test = train_test_split(tfid_df, labels, test_size = 1000)

# Naive Bayes Model
clf2 = MultinomialNB()
clf2.fit(X_train, y_train)
# Scoring the model
clf2.score(X_test, y_test)
# Making predictions on test data
y_pred_test2 = clf2.predict(X_test)
# Making predictions on training data
y_pred_train2 = clf2.predict(X_train)
# Training Accuracy
train_acc2 = accuracy_score(y_train, y_pred_train2)
# Testing Accuracy
test_acc2 = accuracy_score(y_test, y_pred_test2)
```

```
train_acc2
```

```
test_acc2
```

Now I will summarize the results of these two vectorizers, and compare/contrast them.

```
# Compiling dataframe
compare_results = pd.DataFrame({'Train Accuracy': [train_acc, train_acc2],
                                'Test Accuracy': [test_acc, test_acc2]})
compare_results
```

We can see in the compiled dataframe (with 0 being CountVectorizer and 1 being TfidVectorizer), that the accuracies for both the training and testing sets are better with **CountVectorizer**.

In the future, I would also like to optimize the hyperparamters of these vectorizers, but I don't have enough time. Maybe that will be a fun winter-break project :D

By Brianna MacDonald