**Python Book**

1. [Python Fundamentals](#PythonFundamentals)
2. [Data\_Structures](#Data_Structures)
3. [OOP](#OOP)
4. [Strings](#Strings)
5. [Functions](#Functions)
6. [Loops](#Looping)
7. [Sort](#Sorting)
8. [Numpy](#NumpyandSciPy)
9. [SciPy](#SciPy)
10. [Visualizations](#Visualizations)
11. [Data Pre-Proces](#DataPreProcessing)sing
12. [Filtering columns and rows](#Filtering)
13. [Featuretools](#FeatureTools)
14. [PCA](#PCA)
15. [Clustering](#Clustering)
16. [Regression](#Regression)
17. [Classification](#Classification)
18. [Deep Learning](#DeepLearning)
19. [Recommendation Systems](#RecommendationSystems)

**Python Fundamentals** [**@**](#TOC)

**Python Elements**

1. Built in Functions (you can also create your own functions)
   1. is a repeatable procedure
   2. It accepts inputs which are called arguments
   3. Returns a value which is called the function output
   4. You invoke or call a function with a pair of parenthesis
   5. Common functions len, str, int, float, type
   6. Example
      1. first\_name = "Eric"
      2. len(first\_name)
         1. 4
2. Attributes
   1. An **attribute** is a piece of data that lives on an object.
   2. An **attribute** is a fact, a detail, a characteristic of the object.
   3. Access an attribute with object.attribute syntax.
   4. Common attributes – dytpes, shape, values, columns, axes
   5. Examples
      1. me = pd.Series(["smart", "handsome", "humble", "nice", "smart"])
      2. me.values
         1. ["smart", "handsome", "humble", "nice", "smart"]
      3. Me.shape
         1. (5,)
3. Custom Functions
   1. Define a function with “def” keyword
   2. A parameter is the name for an expected input.
   3. Need to use the “return” keyword to specify what output you want
   4. Example
      1. def cel\_to\_fahr(celsius\_temp):
      2. calculation = celsius\_temp \* 1.8 + 32
      3. return calculation
         1. cel\_to\_fahr(0)
            1. 32
4. String Methods
   1. A string is an object
   2. A method is like a function that belongs to an object
   3. To invoke a method, provide a period after the object, then the method name followed by a pair of parentheses
   4. Common string methods – upper, lower, swapcase, title, capitalize, and strip
   5. “in” and “not in” keywords check for the presence in a string
   6. Examples
      1. profession = "Python Developer".upper()
         1. PYTHON DEVELOPER
      2. profession = "Python Developer "
      3. profession.rstrip().lstrip().lower()
         1. 'python developer'
      4. profession.strip().lower().replace("p","")
         1. ‘ython developer’
5. Lists
   1. A list is a mutable data structure that holds an ordered collection of values
   2. Each item in the list is called an element
   3. “append” method adds an element to the end of the list
   4. The “pop” method removes the last element in the list
   5. The “in” and “not in” keywords checks to see if an element exists in the list
   6. Examples
      1. cars = ["Porsche", "Ferrari", "Audi", "BMW"] # a list is an array
      2. cars
         1. ['Porsche', 'Ferrari', 'Audi', 'BMW']
      3. cars.append("Ford")
         1. ['Porsche', 'Ferrari', 'Audi', 'BMW', ‘Ford’]
      4. Cars.pop()
         1. ['Porsche', 'Ferrari', 'Audi', 'BMW'’]
6. Index Positions and Slicing
   1. Every string character has an index position
   2. The index starts at 0
   3. Use square brackets to extract a character/element by index position
   4. Slicing extracts multiple elements from a list
   5. Examples
      1. hero = "Spiderman"
      2. hero[4]
         1. ‘e’
      3. Hero[-1]
         1. ‘n’
      4. Hero[1:3]
         1. ‘pi’
      5. cars = ['Porsche', 'Ferrari', 'Audi']
      6. cars[:2]
         1. ['Porsche', 'Ferrari']
7. Dictionaries
   1. Is a mutable unordered collection of key value pairs
   2. A key is a identifier for a value
   3. A value corresponds with a key (like a series)
   4. Dictionaries require curly brackets { }
   5. And a colon between every key and value
   6. The length of a dictionary is the number of key-value pairs
   7. Example
      1. menu = {"Hamburger": 7.89, "Fries": 2.99, "Small Soda": 1.99, "Medium Soda": 2.29, "Large Soda": 2.49, "Onion Rings": 3.29}
         1. menu["Taco"] = 3.99
         2. menu["Fries"]=3.29 # this changes the price of fries in the dictionary
         3. menu.pop(“Hamburger”) # removes hamburger from the list
         4. ‘Taco’ in menu #boolean statement
            1. True
8. Arrays
   1. Reverse an array

**Data Frames I, II, and III**

**Importing a csv file**

1. pd.read\_csv(“/path/table.csv”)
2. pd.read\_csv("table.csv", parse\_dates=["Date"], date\_format="%m/%d/%Y")
   1. Helps pandas read the incoming date field
3. pd.read\_csv("table.csv").dropna(how="all")
   1. Drop rows with all missing values
4. pd.read\_csv("table.csv", parse\_dates=["date1"], date\_format="%Y-%m-%d", index\_col=["col1", “col2”]).sort\_index()
   1. create an index from cols 1 and 2. The sort index then sorts by 1 first and then
5. Export df to csv
   1. df.to\_csv("df\_output\_name.csv", index=False)
      1. sends df to a csv file.
      2. Index false means that the output will not include the index
6. Import Excel into pandas
   1. Install openpyxl Library to Read and Write Excel Files
   2. import openpyxl as xl
   3. pd.read\_excel("dataset.xlsx")
      1. grabs first tab by default
   4. pd.read\_excel("dateset.xlsx", sheet\_name="data 2")
      1. add sheet name to grab a specific worksheet
7. Export pandas to Excel
   1. import openpyxl as xl to enable Excel exports
   2. df.to\_excel(excel\_file, sheet\_name=”data 2”, index=False)
      1. This exportsdf to tab data2 and it does not include the index
8. Import csv from a website
   1. url = "path/dataset.csv"
   2. df = pd.read\_csv(url) – imports data from a website

**Random Samples**

1. Create random sample
   1. df.sample(n=5) – grabs a random five rows
   2. df.sample(n=2, axis="columns") – can also randomly select columns

**Setting the Index**

* 1. The index serves as the collection of primary identifiers/labels/entrypoints for the rows.
  2. The fastest way to extract a row is from a sorted index by position/label.
  3. Pandas uses index labels/values when merging different objects together.
  4. The set\_index method sets an existing column as the index of the **DataFrame**.
  5. The reset\_index method sets the standard ascending numeric index as the index of the **DataFrame**.
  6. Examples
     1. bond.set\_index("Film") – this changes the index from the autonumber to Film
     2. bond.reset\_index(drop=True) this returns it to the autonumber
     3. bond.reset\_index().set\_index("Year") – this shifts the index from Film to Year

**Describing the df**

1. df.head() or df.tail() to get the top or bottom rows
2. df.dtypes – outputs each variable and its data type
3. df.axes – provides number of rows and column names
4. df.info() – allows you to see all columns, missing values, and dtype
5. df.describe() or df.describe().T for transposed output
   1. provides count, mean, stddev, min/max, and quartiles for every column
6. value\_counts
   * 1. df["col1"].value\_counts()
        1. this provides the count for all values in col1
     2. df["col1"].value\_counts(normalize=True)
        1. to get column percents
     3. You can find the value counts for multiple columns
        1. value\_counts\_df = df[‘col1’,’col2’,’col3’].apply(lambda col: col.value\_counts())
7. df["col1"].nunique()
   1. tells us the number of unique values in the Team column
8. df["col1"]= df["col1"].astype("category")
   1. this converts the Position column from an object dtype to a category dtype. This can save space and make python run faster

**Filtering Data**

1. data.select\_dtypes(include = "number").columns.to\_list()
   * 1. This selects the columns that are numeric
     2. Select types is a method that selects a dtype
     3. Include is the argument and can grab int, float, object, category, datetime, bool
        1. Number includes both int and float
        2. Can also select more than one type just add []
     4. Can also replace include with exclude and filter in reverse
2. df = df["col1"].str.contains("text1") – this selects all col1 values that contain text1
3. df = df["col1"].str.lower().str.endswith("text1") – this selects all col1 values that end with text1
4. df = df["col1"].str.lower().str.startswith("text1") – this selects all col1 values that start with text1
5. df2 = df1[“col1”] – selects one column from the df. The square brackets are used to extract the column from the df
6. df1.iloc[3]=”new value” will change the value in the third row to new value
7. To not permanently change the value use the copy() method
   * 1. df = [‘col1’].copy()
        1. this changes it in memory but not the original tab
8. To select multiple columns
   * 1. df[["col1","col2"]]
        1. The first square brackets mean extraction. The second set are because it is a list
     2. columns\_selected = ["col1","col2"]
        1. you can also create a column list and then call them with
           1. new\_df = old\_df[columns\_selected]
9. Copy a value across all rows
   * 1. df["col\_new"] = "text\_new" – creates col\_new and inserts text\_new across all rows
10. Insert a new column and a value
    * 1. df.insert(loc=3, column="col\_new", value="text\_new")
         1. This adds a column after col 3 and inserts the value text\_new
11. Insert a value and calculate the input
    * 1. df.insert(loc=7, column="col\_2x", value=df["col1"] \* 2)
         1. inserts a field called col\_2x and creates the values by multiplying the existing col1 by 2
      2. To make it permanent you need to add the name and equal sign
         1. df[‘col\_2x’] = df.insert(loc=7, column="col\_2x", value=df["col1"] \* 2)
12. To drop a column
    * 1. df = df.drop(columns=['col\_3x'])
         1. this will drop the column Salary\_3x
13. To create a new column based on a calculation
    * 1. df["col\_new"] = df["col1"] - df["col2"]
14. Boolean Method
    * 1. First, we create a Boolean series - df["col1"] == "text"
      2. Second, we surround the Boolean code with df[] to select the True values
         1. df[df["col1"] == "text"]
15. Isin filter
    * 1. df\_filtered = df[df['col1'].isin(['text1', 'text2'])] – this will give us rows where col1 includes text1 or text2
16. Null and Not Null values
    * 1. df[df["col1"].isnull()]
         1. returns rows where col1 is null
      2. df[df["col1"].notnull()]
      3. df[df["col1"].isnull() & df["col2"].notnull()]
17. Between method
    * 1. df[df["Date"].between("1995-01-01", "1995-12-31")]
18. Duplicate values
    * 1. df.duplicated()
         1. returns the Boolean value
         2. Put brackets around it to remove a duplicate
      2. df[df[“col1”].duplicated(keep=”last”)
         1. This removes the first duplicate value and keeps the second
19. Drop duplicates
    * 1. df.drop\_duplicates("col1")
         1. keeps the first instance and drops the second, which is the default
      2. df.drop\_duplicates( keep=”last)
         1. keeps the last
      3. df.drop\_duplicates(subset=”col1”, keep=”last)
         1. this defines a dup as just col1 and keeps the second instance
      4. df.drop\_duplicates(subset=["col1", "col2"]).sort\_values("col1")
         1. This drops if there is a duplicate in both col1 and col2. Also sorts the df by col1
20. Checking for non digit values in a column
    * 1. df\_digit = pd.DataFrame(df.column.str.isdigit())
         1. This code create a new df that contains the index and a Boolean value from the isdigit() function. Isdigit returns a True if the string is all digits and a False if it is not all digits.
      2. data[hpIsDigit[‘column’] == False]
         1. This use the hpIsDigit df and selects all of the False values, i.e., not digits
21. Iloc accessor
    * 1. Iloc retrieves rows by the index position
      2. Iloc accepts single values, lists, and slices
      3. df.iloc[5] – selects the sixth row because we start at 0
      4. df.iloc[[10, 12]] – this returns rows 10 and 12. The inner brackets are because it is a list and the outer are for the extraction
      5. df.iloc[4:8] – this is the slice. This returns rows 4 through 7
      6. df.iloc[:6] gives the first six rows
      7. df.iloc[20:] returns row 20 until the end
22. loc accessor
    * 1. Define the index when importing the file
         1. df = pd.read\_csv("data.csv", index\_col="col1")
      2. df.loc["text"] – returns the row where index equals “text”
      3. df.loc["text1":"text10"] – this is a between slice
      4. df.loc["text1":] – this is from text1 to the end
      5. df.loc[:"text10"] – this is everything before text10
      6. df.loc[["text1", "text2"], ["col1", "col2"]] this returns the text1 and text2 but only cols 1 and 2

**Isna, Dropna and Fillna – counting, dropping or adding values to missing values**

1. Counts missing values
   1. df.isna() is the boolean
   2. df.isna().sum() counts the missing values for each column
2. Drop rows with missing values
   1. df\_drop\_rows = df.dropna()
      1. removes any row with a ***single*** NaN value
   2. df\_drop\_rows = df.dropna(how=”any”)
      1. this is the default so it is the same as above
   3. df\_drop\_rows = df.dropna(how=”all”) – this drops rows with ***all*** NaN values
   4. Can also specify which columns to use for dropping rows
   5. df.dropna(subset=["col1", "col2"])
      1. this drops if either col is NaN
   6. df.dropna(subset=["col1", "col2"], how="all")
      1. this drops if both are NaN
3. Filling in Missing Values
   * 1. The fillna method replaces NaN with the value of the argument
     2. Fillna returns a copy and ***does not*** override the original data
     3. df\_new = df.fillna(0)
        1. This fills in missing values with a zero
     4. df = df[‘col1’].fillna(df[‘col1’].mean())
        1. This fills in the missing value with the mean for a column
     5. df = df.fillna(df.mean())
        1. This fills in all columns NaN with mean
     6. df["col\_new"] = df["col1"].fillna(value="text")
        1. overwrites the missing value with “text” in only the specified column
4. Filling in missing values for a column but filling them in with means from another dimension. If we have missing values, we can be more targeted when filling these in if we know there is some correlation to another variable.
   1. For example, the number of bathrooms in a home may vary based on geography. Urban homes have fewer while suburban homes have more. Thus, we would not fill in missing values with just one mean but an urban mean and a suburban mean which will be more accurate
      1. df['col1'] = df['col1'].fillna(value = data.groupby(['col2','col3'])['col1'].transform('mean'))
         1. This code fill-in missing values in col1 with means from col2 and col3
         2. The transform function uses the mean to fill in the values

**Changing or restructuring the data**

1. Replacing values
   1. df = df.replace("text1", "text2")
      1. Will replace text1 with text2 across all columns
   2. df["col1"] = df["col1"].replace("text1", "text2").
      1. This replaces text1 with text2 in col1
   3. Rename index labels or columns in df
      1. df = df.rename(columns={"col1":"col1\_new", "col2": "col2\_new"})
2. Transpose method – converts rows to columns and vice versa
3. Stack method moves the column index to the row index
4. Unstack method is the inverse of stack
5. Pivot Method – allows you to create a cross tab query
   1. df.pivot(index="col0", columns="col1", values="col2") – This creates a cross tab query with col0 as the row, col1 as the column, and col2 as the value
6. Melt Method is the inverse of the pivot method
7. df.melt(id\_vars="col1", var\_name="col2", value\_name="col3").sort\_values(["col1", "col2"]) – returns back to df.
8. The Pivot Table Method – \*\*Can turn into a seaborn heatmap\*\*
9. The values parameter accepts the numeric column whose values will be aggregated.
10. The aggfunc parameter declares the aggregation function
11. The index parameter sets the index labels of the pivot table.
12. The columns parameter sets the column labels of the pivot table.
13. df.pivot\_table(values="col1", index=["col2","col3"], aggfunc="sum") – this creates a table with col2 and col3 as rows, col1 as column and the values are summed. The aggfunc parameter can also accept mean, count, min, and max
14. Turn pivot into a heatmap
    * df\_man = df[df.borough == 'Manhattan']
    * df\_hm = df\_man.pivot\_table(index = 'start\_hour', columns = 'week\_day', values = 'pickups')
    * # Draw a heatmap
    * plt.figure(figsize = (20, 10)) # To resize the plot
    * sns.heatmap(df\_hm,  fmt = "d", cmap = 'coolwarm', linewidths = .5, vmin = 0)
    * plt.show()

**Formats**

1. astype method
   1. df["col1"] = df["col1"].astype("int")
      1. this changes col1 from its current format to an integer
   2. df["col1"] = df["col1"].astype("category")
      1. astype category method. Changes an object or integer to category which saves space. Reserve this for columns with minimal variation
   3. df["col1"] = df["col1"].astype("float") – converts to float
2. Dates
   1. df["col1"] = pd.to\_datetime(df["col1"], format="%m/%d/%Y") – this changes “col1” from a text string to a date field
   2. df["col1"] = pd.to\_datetime(df["col1"], format="%H:%M %p").dt.time – this converts string into time
   3. Once the column is formatted as a date, we can format it how we would like
      1. df[“col1”].dt.strftime(“%m/%d/%Y) - this will give us 04/25/2024
      2. df[“year”] = df[“col1”].dt.year – this will give us just the year
      3. df[“month”] = df[“col1”].dt.month – this will give us just the year

**Sorting data**

1. Sort a df by the values
   1. df.sort\_values(by="col1", ascending=False)
      1. this sorts col1 in descending order
   2. df.sort\_values(by="col1", ascending=False, na\_position="last")
      1. the na\_position parameter sorts NaN at the bottom of the list
   3. df.sort\_values(by=["Position", "Salary"], ascending=[False, False])
      1. this is sorted by two columns and both are descending
2. Sort a df by the index
   * 1. nba.sort\_index(ascending=False) – this sorts the index value in descending order
3. Rank values – this method assigns a numeric ranking to each series value
   * 1. nba["Salary Rank"] = nba["Salary"].rank(ascending=False).astype(int) – this ranks salaries based on the descending sort

**Working with Text Data**

1. df["col1"] = df["col1"].str.title().str.strip()
   1. This code capitalizes the column position title and strips out blank spaces
2. df["col1"] = df["col1"].str.title().str.strip().str.replace("text1", "text2")
   1. does the same as above but also changes text1 to text2
3. df["col1"] = df["col1"].str.title().str.replace("$", "").str.strip()
   1. does all the above but also replaces $ with a blank and then strips out all blanks
4. Filtering text
   1. df = df["col1"].str.contains("text1") – this selects all col1 values that contain text1
   2. df = df["col1"].str.lower().str.endswith("text1") – this selects all col1 values that end with text1
   3. df = df["col1"].str.lower().str.startswith("text1") – this selects all col1 values that start with text1
5. Split text based on a delimiter
   1. df["col1"].str.split(" ") – splits text wherever there is a space

**groupby**

1. df.groupby(“Col1”)[“Col5”].mean()
   1. The group by method starts by defining what col to group on and then what col to perform the mean, median, min, etc.
2. df.groupby([“Col1”, “Col2”])[[“Col4”, “Col5”]].mean()
   1. If multiple columns are used for either the grouping or to find the mean, then square brackets are introduced because we are calling a list of columns
3. df.groupby(['Col1'])['Col2'].agg(table\_label1='unique', table\_label2='count')
   1. Based on the group of Col1 and defining Col2 as the column to perform the math, you can add agg and then define the unique values from Col2 and then count the rows from the groupby Col1 and Col2
   2. This is very powerful because it allows you to see multiple statistics, from unique, median, mean, etc associated with grouped columns

**Merging df’s**

1. Merge and concat
   1. Merge is a method so there is no pd
      1. Df1.merge(df2, how=”inner”, on=”key”)
      2. Df1.merge(df2, how=”inner”, right\_on=”key1”, left\_on=”key2”)
      3. Df1.merge(df2, how=”inner”, right\_on=”key1”, left\_on=”key2”).drop("key2", axis="columns")
   2. Concat is a function so there is a pd
      1. pd.concat([df1, df2])
      2. pd.concat([df1, df2], ignore\_index=True)
2. Concat can stack two dfs on top of one another or side by side. Concat does not join on key values, however. Concat makes sense when the two dfs have the same common keys or columns
3. Concat method stacks dataset
   1. pd.concat([df1, df2]) #concat stacks the two dataframes. Same field names makes it easy
   2. pd.concat([df1, df2], ignore\_index=True)
      1. this creates a new index. The two df had 250 rows and when stacked we have an index that goes from 0 to 249 with each stacked table.
      2. If you ignore\_index=True, it will create a new index from 0 to 499.
   3. pd.concat([df1, df2], keys=["col1", "col2"]).sort\_index()
      1. Note the keys in this example are not keys like in SQL or a merge. The keys parameter creates a hierarchical index to differentiate between the stacked DataFrames but not to align the tables based on a key field or column
   4. pd.concat([df1,df2], axis="index") – this merges the two dfs. If the index values are equal, it will join those rows. If they are not equal, a new row will be created. It is like a union join. All rows get merged together whether they are equal or not
4. The merge method
   1. df1.merge(df2, how="left")
      1. left join gives you all of the values from df1 and appends data from df2 where the field name values are equal.
      2. Because we do not specify a column to join on, it defaults to index in each table
   2. df1.merge(df2, how="left", on=”key”)
      1. both df’s have the same col name for the join
   3. df1.merge(df2, how="left", left\_on="col1", right\_on="col2").drop("ID", axis="columns")
      1. col1 and col2 have different names so they need to be defined with left\_on and right\_on
      2. The ID name can be dropped because python will print out both. The axis command is used to tell python that we are dropping a column.
   4. Inner join multiple columns
      1. df1.merge(df2, how="inner", on=["col1","col2"])
   5. Outer join
      1. df1.merge(df2, how="outer", on=["col1"])
   6. Use indexes to join
      1. df1.merge(df2, how="left", left\_on="col0", right\_index=True)
         1. in this case the index is col0 so you can use it to join the sales data
         2. right\_index or left\_index simply says the index comes from the left or right df

**Chaining**

Chaining combines several commands into one bit of code

#first we filter on brands with gt 10k miles and then we sort the remaining values

top\_brands = data[data['Kilometers\_Driven'] > 10000][['brand', 'Kilometers\_Driven']].sort\_values(by='Kilometers\_Driven', ascending=False)

#same code written more clean

avg\_miles\_brand = (

data[data['Kilometers\_Driven'] > 10000]

.groupby('brand')['Kilometers\_Driven']

.mean()

.reset\_index() )

# Chaining to clean and transform data

cleaned\_data = (

data .drop\_duplicates()

.fillna({'Kilometers\_Driven': 0})

.rename(columns={'Kilometers\_Driven': 'km\_driven'})

.astype({'km\_driven': 'int'}) )

#Chaining to change age into age group

data['age\_group'] = (

data['age']

.apply(lambda x: 'young' if x < 30 else 'middle-aged' if x < 60 else 'senior') )

#Chaining to group, sum, and then sort in descending order

avg\_sales = (

sales\_data

.groupby(['region', 'product'])['sales']

.sum()

.reset\_index()

.sort\_values(by='sales', ascending=False) )

# Chain to merge multiple tables at once

combined\_data = (

df1

.merge(df2, on='key')

.merge(df3, on='key')

.dropna() )

**[Data Structures](#Data_Structures)** [**@**](#TOC)

Lists

Access list via index:

* list[0] will return the first element in the list
* list[1,3] will slice the second and third element in the list.
* list.append(‘text’) will add ‘text’ on the end of the list
* del list[2] will delete the third item in the list
* list.remove(‘text’) will remove the item from the list where text=’text’
* for list\_value in list: allows us to interate over the list
* if ‘text’ in list: allows us to to check for inclusion
* enumerate can be used to capture both the list index and list value
* Making changes to the original list
  + Must use an index to make changes to original list
  + If we use the temp list values, it will change the temp value but not the values in the list. The example below will only change the temp list value but not change the list
    - for num in numbers:
      * num -= 20
  + We must iterate over the index and not the list values to make changes directly to the list. See example below and how it differs from the example above
    - for i in range(len(numbers)):
      * num[i] -=20
* Keeping the values the same in the original list and creating a new list
  + We must initialize the new list and then use the append function. This code will do the same as the above code but creates a new list
    - new\_list = [ ]
    - for num in numbers:
      * new\_list.append(num-20)
* Add numbers from two lists with the same number of elements and append to a new list
  + Need to interate over index because we need to use num[i]
  + Since we have the same number of elements [i] will word for both lists
    - New\_list = [ ]
    - For i in range(len(numbers)):
      * New\_list.append(num\_list1[i] + num\_list2[i])
* Add numbers from two lists with a different number of elements
  + Initialize a new list new\_list = [ ]
  + We want to iterate over the shorter list
  + So if list1 has 5 elements and list2 has 7, we will only use the first five items from both lists
  + We will need to append items 6 and 7 from list two to the new list
* We can also merge two lists
  + The simple way is new\_list = list1 + list2
  + If order matters, things become more complex
  + i, j = 0, 0
  + new\_dict = [ ]
  + while i < len(list1) and j < len(list2):
    - if list1[i] < list2[i]:
      * new\_dict.append(list1[ i ])
      * i += 1
    - if list1[i] > list2[i]:
      * new\_dict.append(list2[ j ]
      * j += 1
  + new\_dict += list1[ i: ] these last two line append any remaining values
  + new\_dict += list2[ j: ]
* We can also find the intersection of two lists
  + i, j = 0, 0
  + new\_dict = [ ]
  + while i < len(list1) and j < len(list2):
    - if list1[i] < list2[i]:
      * i += 1
    - elif list1[i] > list2[i]:
      * j += 1
    - else:
      * new\_dict.append(list1[i])
      * i += 1
      * j += 1

Tuples

* Tuple values cannot be changed. If you want to change tuple values, you must unpack and then repack into a new tuple
* tuple[0] will return the first element in the tuple
* tuple[1,3] will slice the second and third element in the tuple.
* for tuple\_value in tuple: allows us to interate over the tuple
* if ‘text’ in tuple: allows us to to check for inclusion
* we can also unpack a tuple
  + element 1, element 2, element 3 = tuple
  + This creates three new temp variables
* Tuples can be used in lists
* List = [(1,2), (3.4), (5.6)]
  + The first tuple (1,2) is the first item in list so list[0]==(1,2)
  + To access a tuple value, use the double index. To access the 3 the index would be list[1][0]. The first [] selects the list position and the second [] selects the tuple location
  + del list[x] will delete a tuple from a list as will list.remove(num)
* The zip function will zip two lists and create tuples for each combination
  + If the lists are the same size, it will zip all values and if one list is smaller it will only zip for those values that overlap.
  + List1 = [1, 2, 3]
  + List2 = [4, 5, 6]
  + For num in zip(list1, list2):
    - Print(num) would return three individual tuples
    - (1, 4)
    - (2, 5)
    - (3, 6)

Dictionary

* A dictionary is like a lookup table. It has unique key value pairs
* To access a value it is the dictionary name and the key
  + dictionary[‘key’] will return the value
* To update a value
  + Dictionary[‘key’] = new\_value
* Add a value
  + dictionary[‘new\_key\_value’] = value
  + Because new\_key\_value is not in the dictionary, it will be added as the key and then whatever is on the other side of the equal sign will be the value
* Delete a dictionary value
  + Del dictionary[‘key’]
* Iterating over a dictionary. Cannot add or remove a value when iterative over it
  + When using keys we must use the dictionary name and [key] in the sum line so it knows we want to sum the values. dictionary[key] returns the value
  + for key in dictionary.keys()
    - for key in dictionary.keys():
      * Sum += dictionary[key]
  + For value in dictionary.values():
  + In this example we do not need to reference the key because we are explicitely using the dictionary values
    - For value in dictionary.values():
      * Sum += values
  + For key, value in dictionary.items():
  + In this example we want to use both the key and value. Once we define the max value or max key, the other is also associated so once we know one, we know the other. In the example below, once we know the max value, we can also return the key associated with the max value
  + For key, value in dictionary.items():
    - If value>max:
      * max\_value=value
      * max\_key=key
* Modifying a dictionary value
  + With lists, we can change a value by referencing its index. When changing the values of the dictionary, we will need to reference the key.
  + For key, value in dictionary.items():
    - If x>0:
      * Dictionary[key]= value + 10
* Counting list elements and creating a dictionary
  + This version takes a list and turns it into a dictionary. The individual elements of the list become the key and the count of elements becomes the values
  + List = [a, b, c, a, b, a]
  + New\_dict = {}
  + For letter in list:
    - If letter not in new dict: #this is a new list so the letter will not be their
      * New\_dict[letter] = 0 #adds the letter to the list as the key and a
    - New\_dict[letter] += 1 #increments by one for each element in key
  + Output would be new\_dict{‘a’ : 3, ’b’ : 2, ’c’ : 1}
* Counting elements in tuples with dictionaries
  + List = [(john, volvo), (bob,ford), (kate,volvo)]
  + New\_dict = {}
  + For people in list:
    - name, car\_make = people #unpacking the tuple
    - If car\_make not in new\_dict:
      * new\_dict[car\_make] = 0 #add the car make to the new dict
    - new\_dict[car\_make] += 1 #increment each car make vote
    - Output would be new\_dict = {‘volvo’: 2, ‘ford’: 1}
* The get() function
  + The get function simplifies these three lines of code
    - If car\_make not in new\_dict:
      * new\_dict[car\_make] = 0 #add the car make to the new dict
    - new\_dict[car\_make] += 1 #increment each car make vote
  + It is replace with the code
    - New\_dict[car\_make] = new\_dict.get(car\_make, 0) + 1
    - car\_make is the key for new dict. 0 is first applied and then incremented by 1 each time the key is found
    - new\_dict[key] =new\_dict.get(key,initial value) + increment by value
* Tuples can also be the dictionary key
  + In this example, we are rolling two dice and capturing the values as tuples, eg., dice\_rolls = [(3,5), (4,2), (6,1)]
    - New\_dict = {}
    - For roll in dice\_throws:
      * New\_dict[roll] = results.get(roll,0)+1
    - The output is the roll combination tuple, and the value is the number of times that roll happened
* Grouping with Dictionaries
  + We can group keys based on the number of characters in a name or we can group airports based on the number of runways.
  + The trick is groups[key] = [ ]. This line creates the key which is the group value and then we append values, usually strings as the values. The blank list allows us to append multiple strings
    - Airport\_runways = [(‘Warsay’, 2), (‘Vienna’, 2), (‘Frankfurt’, 4)]
    - runways = {}
    - For airport in airport\_runways:
      * City, runway\_count = airport
      * If runway\_count not in runways:
        + runways[runway\_count] = [ ]
      * runways[runway\_count].append(city)
    - The output uses the runway count as the key and then the airport names are caputed in a list which are the dictionary values
* We can use get() with grouping
  + This code does the same as the code above
    - runways = {}
    - For airport in airport\_runways:
      * City, runway\_count = airport
      * Runways[runway] = runways.get(runway, [ ]) + [airport\_name]
    - For the interation, the + [airport\_name] appends each airport name based on the number of runways
* We can also use the update function to link dictionaries
  + We often link using the update function to update values in a dictionary
    - Salaries\_2019 = {‘Anna’: 5000, ‘Bob’: 4000, ‘Tim’: 4500}
    - Salaries\_update = {‘Anna’: 6000, ‘Bob’: 4500}
    - Salaries\_2019.update(salaries\_update)
    - This one line of code will update salaries\_2019 with the higher salaries for Anna and Bob
* Linking dictionaries which allows you to define default values
  + Create a dictionary with defaults
  + Create a copy of the original dictionary
  + Use the update function to update the dictionary copy
  + For those elements not updated, it will use the defaults which remain from when it was copied

Sets

* Sets are an unordered collection of elements. The elements must be unique so there can be no duplicates
* Add an element set.add(‘text’)
* Remove an element set.discard(‘text’)
* Remove all elements set.clear()
* Iterate over for num in numbers:
* If num in numbers for conditional code
* A.intersection(B) or A & B
  + This returns where the two sets overlap
* A.union(B) or A | B
  + Returns all elements of both lists
* A.difference(B) or A – B
  + Returns elements in A and not in B
* A.symmetric\_difference
  + Returns elements in A and not in B and elements in B not in A
* A.issubset(B) or A<=B
  + Returns true or false
* A.superset(B)
  + Returns true or false

**[OOP](#OOP)** [**@**](#TOC)

**Fundamentals**

* **Class** is a blueprint or template. It defines what an object is (its attributes) and what it can do (its methods). It defines a set of attributes (data) and methods (functions) that the create objects (instances) will have. If we had a dog class, the template would include the dog name and dog type. They would essentially be placeholders for values that would be included when the object calls the class.
* **Object** is an instance of a class. A concrete realization of the class with specific value for its attributes. It has its own data. If we had a dog class, the object data would be the dog name and type of dog (Rufus, Saint Bernard)
* **Attributes** are properties and data. Characteristics that describe the object(color, size). The attribute is a variable that holds the data assocated with the class or an object. Attributes hold data that define the state of an object. A class attribute is shared by all instances of the class while an instance attribute is unique to each object instance
* **Method** is a function defined within a class that describes the behaviors or actions that objects or the class can perform. Actions the object can perform or that can be performed on the object.
* **Constructor (\_\_init\_\_ Method)** is a special method used to initialize new objects of a class. It sets up the initial state of an object by assigning values to its attributes. This initial state is the blueprint.
* **self Keyword** represents the instance of the class. It allows access to the attributes and methods of the class in Python. self is passed automatically to instance methods and is used to differentiate between instance attributes and local variables.

**OOP Principles**

* **Inheritance** allows a new class(child or subclass) to inherit attributes and methods from an existing class (parent or superclass)
* **Encapsulation** is the bundling of data (attributes) and methods that operate on that data within a single unit (class). It restricts direct access to some of the object components, which can prevent accidental interference or misuse
* **Polymorphism** allows objects of different classes to be treated as objects of a common superclass. It enables the same operation to behave differently on different classes.
* **Abstraction** is the concept of hiding complex implementation details and explosing only the necessary parts of an object. It simplifies the interaction of objects by providing a clear interface.

**Understand the Relationship between classes**

* **Composition** is a design principle where a class is composed of one or more objects of other classes, implying as “has-a” relationship. It promotes modularity and code reuse.
* **Aggregation** is a form of association that represents a “whole-part” relationship between classes, where the part can exist independently of the whole. It is a weaker form of composition.
* **Association** defines a relationship between two classes, where one class uses or interacts with another. It represents a “uses-a” relationship.

**Strings**

**Convert a number to a string**

* str(integer)

**Input function**

* user\_input = input(“Please type in your first name”)

**Join function**

The separators can be a “” or any text or string. The elements can be any data structure. So you can

* path\_parts = ['c:','users','john','images','camera','edited','001.img']
* print('\\'.join(path\_parts))
* c:\users\join\images\camera\edited\001.img

**in operator**

It checks wheter a text character is present in a string

* If ‘@’ not in email\_address:
  + Print(“please enter a real email address”)

**len() function**

* len(‘text’)
* Will return the number of letters in the text or list or tuple

**Index() and find() functions**

* Index(‘text’) and find(‘text’)
* It will return the index where the text first occurs
* Find will return a -1 if the text is not in the string being searched and index will raise a ValueError

**Count() function**

* Sentence.count(‘a’)
* Will return the number of a’s in the sentence

**Is…() functions**

* The functions return a True or False and are often used in conditional statements
* str.Isdigit() returns True if a digit is in the string
* str.is alpha() returns True if all characters in the string are letters
* str.isalnum() returns True if all characters in the string are letters or digits
* str.isspace() returns True if all characters in the string are whitespaces
* str.islower() or str.isupper() returns True if all chacters in the string are upper or lower case

**startswith() and endswith() functions**

* string.startswith(‘letter’) or string.endswith(‘letter’). Both return a True if the argument is true

**upper(), lower(), and capitalize() functions**

* string.upper() will convert the first letter to a capital
* string.lower() will convert all letters in the string to lower case
* string.upper() will capitalize all letters in the string

**strip(), lstrip(), and rstrip() functions**

* string.strip(‘,’) will remove commas from the string
* string.strip() will remove whitespaces at the beginning or end of the string
* lstrip() removes characters from the beginning of the string
* rstrip() remove characters from the end of the string

**replace() function**

* string.replace(‘a’, ‘b’)
* This would replace all a’s with b’s. Can also replace words

**Split() function**

* String.split(separator)
* Email\_address.split(‘@’) will split the email by the @, returning two substrings

**Functions** [**@**](#TOC)

**Function Parameters**

1. Positional Parameters - The values you pass to the function must be in the same order as the parameters are defined. When calling greet("Alice", 30), "Alice" is matched to name, and 30 is matched to age by position.

def greet(name, age):

    print(f"{name} is {age} years old.")

2. Default Parameters - In this example, if no age is given in the function call, 25 is entered. Calling greet("Alice") will assume age is 25, while greet("Alice", 30) overrides the default.

def greet(name, age=25):

    print(f"{name} is {age} years old.")

3.  Keyword only parameters - To force a parameter to be passed by keyword rather than position, you can define it after a \* symbol. This is useful for optional parameters when you want to ensure clarity. Here, greet("Alice", age=30) works, but greet("Alice", 30) would cause an error, as age must be explicitly named.

def greet(name, \*, age=25):

    print(f"{name} is {age} years old.")

4.  Variable positioning - Use *\*args to accept any number of positional arguments. These are packed into a tuple, allowing the function to handle varying numbers of inputs. Calling add\_numbers(1, 2, 3, 4) will sum all four numbers, and add\_numbers(5, 10) will sum two numbers. This makes \**args flexible for many inputs.

def add\_numbers(\*args):

    return sum(args)

5. Variable Keyword Patterns - With \*\*kwargs, you can accept any number of keyword arguments, packed into a dictionary. Calling build\_profile("Alice", age=30, city="NY") adds age and city to the profile dictionary. This is useful for handling dynamic, named information.

def build\_profile(name, \*\*kwargs):

    profile = {"name": name}

    profile.update(kwargs)

    return profile

6. Positional only parameters - You can enforce certain parameters to be positional only by placing them before a / in the parameter list. This is less common but is useful in some scenarios to avoid confusion with keywords. In this example, a and b must be passed positionally, while c can be passed either positionally or as a keyword.

def example(a, b=2, *\*args, key1="default", key2="default2", \**\*kwargs):

    print(f"a: {a}, b: {b}")

    print(f"args: {args}")

    print(f"key1: {key1}, key2: {key2}")

    print(f"kwargs: {kwargs}")

example(1, 3, 4, 5, key1="value", extra="info")

# Output:

# a: 1, b: 3

# args: (4, 5)

# key1: value, key2: default2

# kwargs: {'extra': 'info'}

**Lambda function simplifies the function syntax**

**df['price\_with\_tax'] = df['price'].apply(lambda x: x \* 1.10)**

* This applies a 10% tax to every value in price and is saved in price with tax
* X is defined with the code before .apply. Thus we have defined x to be df[‘price’]

**filtered\_df = df[df['age'].apply(lambda x: x > 30)]**

* This filters the df by age, only keeping age gt 30

**df['grade'] = df['score'].apply(lambda x: 'Pass' if x >= 60 else 'Fail')**

* This inserts Pass if score is gte to 60 and Fail if below

**df['squared'] = df['numbers'].apply(lambda x: x \*\* 2)**

* This squares values in numbers column and saves them in squared

**df['full\_name'] = df.apply(lambda x: f"{x['first\_name']} {x['last\_name']}", axis=1)**

* This combines the first and last name columns
* The x represents each row when using it with apply() with axis=1. This function operates row-wise, so for each interation, x is a series object reprensenting the current row
* In this example, think of x as each row, we can access all of the columns in that row. So we can combine the first and last name columns
* Thus, as the function goes through each row, x increases by one each iteration. So we will grab the firstname from row 1 and the lastname of row 1
* Think of x as a proxy for the row

**df['total\_score'] = df.apply(lambda x: x['math'] + x['science'] + x['english'], axis=1)**

* This accesses row one at a time and pulls the math, science, and english scores and then adds them

**df['status'] = df.apply(lambda x: 'Pass' if (x['math'] + x['science'] + x['english']) / 3 > 75 else 'Fail', axis=1)**

* This accesses each row one at a time, averages the scores and then applies else if logic to pass if greater than 75 and fail if lower
* axis = 1 means the lambda function is reading one row at a time

**df['category'] = df.apply(lambda x: 'Premium' if x['price'] > 1000 and x['in\_stock'] else 'Affordable' if x['in\_stock'] else 'Out of Stock', axis=1)**

* For the if then else the first label is listed, then the condition and then the else condition

**df['level'] = df.apply( lambda x: 'Junior' if x['experience'] < 5**

else 'Mid' if x['experience'] < 20 and x['age'] < 40

else 'Senior' if x['experience'] >= 20 and x['age'] < 65

else 'Retired', axis=1 )

* Lambda allows you to do compound if then else statements

**df['reversed\_rating'] = df['rating'].apply(lambda x: x - 6)**

* Use this to recode a column from 5 to 1 to 1 to 5
* This makes a 1 a 5, 2 a 4, 3 a 3, etc.

**Loops** [**@**](#TOC)

Count +=1 means count=count+1

Total += num means Total = Total + num

x -= y means x = x – y

for all of these, they update the value of the variable by performing an operation on it and then assigning the result back to the variable as it loops

**1) Loop basics**

fruits = ['apple', 'banana', 'cherry']

for fruit in fruits:

print(fruit)

Output:

apple

banana

cherry

**2) using range with a for loop**

for i in range(5):

print(i)

Output:

0

1

2

3

4

5

The loop returns each number because the print in within the loop

**3) Math loops**

numbers = [1, 2, 3, 4, 5]

total = 0

for num in numbers:

total += num

total

Output:

15

The loop only returns 15 because total is outside the loop

If total was within the loop it would output each row

0

3

6

10

15

**The looping process**

Total = total + num

0 = 0 + 1 first pass

3 = 1 + 2 second loop

6 = 3 + 3 third loop

10 = 6 + 4 fourth loop

15 = 10 + 5 final loop

**4) While loops**

count = 0

while count < 5:

print(count)

count += 1

Output:

0

1

2

3

4

For while loops, you place the counter at the end.

Looping process – python first checks the while logic, if true it moves to the next line, if false the loop stops. Thus is count is lt 5, then print(count) and then increment

Count = count + 1

0 # the zero is the first output because the print(count) comes before the count+=1, and it is initialized to 0

0 + 1 = 1

1 + 1 = 2

2 + 1 = 3

3 + 1 = 4

**4) Conditional loops**

numbers = [1, 2, 3, 4, 5]

for num in numbers:

if num % 2 == 0:

print(f"{num} is even")

else:

print(f"{num} is odd")

Output:

1 is odd – 1 is not divisable by 2 so num % 2 is false and the else print executes

2 is even

3 is odd

4 is even

5 is odd

In this example there is no counter, but python knows to jump to the next item in the list once the conditional statement has completed because the conditional statement is within the loop

**5) Breaking out of loops**

for i in range(10):

if i == 5:

break

print(i)

Output:

0

1

2

3

4

As soon as I = 5 the loop stops

**6) Continue loop to exclude some values in the loop**

for i in range(5):

if i == 2:

continue

print(i)

Output:

0

1

3

4

2 is skipped because the print statement is not activited. Once python ready continue, it immediately pulls the next number in the range without moving to print(i)

**7) Nested loops**

**Example 1**

i = [1, 2, 3]

j = [1, 2, 3]

for i in range(1, 4): # Outer loop which iterates over the values 1, 2, and 3

for j in range(1, 4): # Inner loop which iterates over 1, 2, and 3 for each i

print(f"{i} \* {j} = {i \* j}")

Output:

1 \* 1 = 1

1 \* 2 = 2

1 \* 3 = 3

2 \* 1 = 2

2 \* 2 = 4

2 \* 3 = 6

3 \* 1 = 3

3 \* 2 = 6

3 \* 3 = 9

Looping process

The outer loop sets i = 1 to represent the first row

Start inner loop (j = 1 – 3)

I \* j = ij this is also outer \* inner = ij

1 \* 1 = 1

1 \* 2 = 2

1 \* 3 = 3

2 \* 1 = 2

2 \* 2 = 4

2 \* 3 = 6

3 \* 1 = 3

3 \* 2 = 6

3 \* 3 = 9

**Example 8**

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for row in matrix:

for num in row:

print(num, end=' ')

row represents each row in the matrix above

num represents each number within each row

**loop process**

Outer loop selects the first row [1, 2, 3] and then python moves to the inner loop which will iterate across the first row to print and create a space

Output for first row: 1 2 3

Then out loop select the second row [4, 5, 6] and then python moves to the inner loop to print and create a space

Output after the second: 1 2 3 4 5 6

Then out loop select the third row [7, 8, 9] and then python moves to the inner loop to print and create a space

Output for third: 1 2 3 4 5 6 7 8 9

* **Outer Loop:** Iterates over the matrix, assigning each sublist to row.
* **Inner Loop:** Iterates over elements within each row, assigning each element to num.
* The print(num, end=' ') statement prints each number on the same line, with a space separating them (end=' ').

**Example**

# Multiplication table (1 to 5)

for i in range(1, 4): # Outer loop for rows

for j in range(1, 4): # Inner loop for columns

print(f"{i \* j:2}", end=' ') # Print product with spacing

print() # Move to the next line after inner loop

Outer loop iterates from 1 to 3

Inner loop iterates from 1 to 3 for each 1

i \* j prints the product for each combination.

The print() after the inner loop moves the cursor to a new line, creating the tabular format.

**Loop Process**

Outer loop (i=1)

Inner loop (j is 1- 3)

1 \* 1 = 1

1 \* 2 = 2

1 \* 3 = 3

Output(1, 1-3) = 1 2 3

Outer loop (i=2)

Inner loop (1-3)

2 \* 1 = 2

2 \* 2 = 4

2 \* 3 = 6

Output(2, 1-3) = 2 4 6

Outer loop (i=3)

Inner loop (1-3)

3 \* 1 = 3

3 \* 2 = 6

3 \* 3 = 9

Output(2, 1-3) = 3 6 9

Output(1-3, 1-3)

1 2 3

2 4 6

3 6 9

**Example 4**

# Define two matrices

matrix1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

matrix2 = [[9, 8, 7], [6, 5, 4], [3, 2, 1]]

# Result matrix

result = [] # this is a list of lists

# Outer loop to iterate over rows

for i in range(len(matrix1)):

row = [] # Create a new row for the result matrix!!!

# Inner loop to iterate over columns

for j in range(len(matrix1[0])): #this calculates the number of elements in the first row which is how many times j will loop

row.append(matrix1[i][j] + matrix2[i][j]) # Add corresponding elements

result.append(row) # Add the completed row to the result

# Print the result matrix

for row in result:

print(row)

loop process

matrix1[i][j] = matrix1[1][1] = 1

matrix2[i][j] = matrix2[1][1] = 9

matrix1[i][j] + matrix2[i][j] = 10

[10]

matrix1[i][j] = matrix1[1][2] = 2

matrix2[i][j] = matrix2[1][2] = 8

matrix1[i][j] + matrix2[i][j] = 10

[10, 10]

matrix1[i][j] = matrix1[1][3] = 3

matrix2[i][j] = matrix2[1][3] = 7

matrix1[i][j] + matrix2[i][j] = 10

[10, 10, 10]

matrix1[i][j] = matrix1[2][1] = 4

matrix2[i][j] = matrix2[2][1] = 6

matrix1[i][j] + matrix2[i][j] = 10

[10, 10, 10]

[10]

matrix1[i][j] = matrix1[2][2] = 5

matrix2[i][j] = matrix2[2][2] = 5

matrix1[i][j] + matrix2[i][j] = 10

[10, 10, 10]

[10, 10]

matrix1[i][j] = matrix1[2][3] = 6

matrix2[i][j] = matrix2[2][3] = 4

matrix1[i][j] + matrix2[i][j] = 10

[10, 10, 10]

[10, 10, 10]

Same for the third round and we end up with

result = [[10, 10, 10], [10, 10, 10], [10, 10, 10]]

Which is the same as

[10, 10, 10]

[10, 10, 10]

[10, 10, 10]

* result.append(row) creates a row individually [10, 10. 10]. This is a list of individual items
* Each row once completed then loaded into result which is a list of lists

Example 5

vowels = 'aeiouAEIOU'

string = "Data Science"

count=0

for letters in string:

  for vowel in vowels:

    if letters == vowel:

      count = count + 1

print(count)

loop process

(i=D, j=a to U) not equal

(i=a, j= a to U) equal to count=1

(i=t, j= a to U) not equal

(i=a, j= a to U) equal so count goes to 2

(i=S, j= a to U) not equal

(i=c, j= a to U) not equal

(i=i, j= a to U) equal so count goes to 3

(i=e, j= a to U) equal so count goes to 4

(i=n, j= a to U) not equal

(i=c, j= a to U) not equal

(i=e, j= a to U) is equal so count goes to 5

**9) Accumulating results in a loop**

numbers = [1, 2, 3, 4, 5, 6]

even\_numbers = [] #when appending you need to initialize the list before the loop

for num in numbers:

if num % 2 == 0:

even\_numbers.append(num)

print(even\_numbers)

Output:

[2, 4, 6]

Loop process

1 is not divisable by 2 so loop does not move to even\_numbers.append(num)

2 is divisable by 2 so the loop moves to even\_numbers.append(num) and appends 2 into the even\_numbers list

3 is not divisable by 2 so loop does not move to even\_numbers.append(num)

4 is divisable by 2 so the loop moves to even\_numbers.append(num) and appends 4 into the even\_numbers list

5 is not divisable by 2 so loop does not move to even\_numbers.append(num)

6 is divisable by 2 so the loop moves to even\_numbers.append(num) and appends 6 into the even\_numbers list

**Example**

* + - 1. We have a table with car models and mile driven. Create a loop which calculates the miles driven by brand and then outputs it to another table.

# Create unique list of brands for the loop

brands = data['brand'].unique().tolist()

# Create an empty list to store results

avg\_miles\_list = []

# Loop through each brand

for brand in brands:

# Filter the DataFrame for the current brand and calculate the average miles

avg\_miles = data[data['brand'] == brand]['Kilometers\_Driven'].mean()

# Append the result to the list as a dictionary

avg\_miles\_list.append({'brand': brand, 'average\_miles': avg\_miles})

# Convert the list to a DataFrame

avg\_miles\_brand = pd.DataFrame(avg\_miles\_list)

# Display the result

print(avg\_miles\_brand)

**Sorting** [**@**](#TOC)

Df[‘col1’].rank()

**Numpy and SciPy Notes** [**@**](#TOC)

**Create a list pandas**

arr\_str = ['Mercedes', 'BMW', 'Audi', 'Ferrari', 'Tesla']

**Convert the list arr\_str to a NumPy array**

np\_arr\_str = np.array(arr\_str)

**Display lists in a matrix format. Matrix is aka a multidimensional array**

matrix = np.array([[1,2,1],[4,5,9],[1,8,9]])

print(matrix)

[[1 2 1]

[4 5 9]

[1 8 9]]

**Create a list of 10 elements. Start and stop are the first two arguments**

arr2  = np.arrange(start = 0, stop = 10)

print(arr2)

[0 1 2 3 4 5 6 7 8 9]

**The third argument is step**

arr3  = np.arrange(start = 0, stop = 21, step = 5)

arr3

[ 0, 5, 10, 15, 20]

**Linespace() provides the values within the start and stop arguments. The default is 50 elements. The formula is (stop - start) / (total elements - 1)**

matrix2 = np.linespace(0,5) # by default 50 evenly spaced values will be generated between 0 and 5

print(matrix2)

[0. 0.10204082 0.20408163 0.30612245 0.40816327 0.51020408

0.6122449 0.71428571 0.81632653 0.91836735 1.02040816 1.12244898

1.2244898 1.32653061 1.42857143 1.53061224 1.63265306 1.73469388

1.83673469 1.93877551 2.04081633 2.14285714 2.24489796 2.34693878

2.44897959 2.55102041 2.65306122 2.75510204 2.85714286 2.95918367

3.06122449 3.16326531 3.26530612 3.36734694 3.46938776 3.57142857

3.67346939 3.7755102 3.87755102 3.97959184 4.08163265 4.18367347

4.28571429 4.3877551 4.48979592 4.59183673 4.69387755 4.79591837

4.89795918 5. ]

**Generate 10 evenly spaced values between 10 and 20**

matrix3 = np.linspace(10,101,10)

print(matrix3)

[ 10. 20.11111111 30.22222222 40.33333333 50.44444444

60.55555556 70.66666667 80.77777778 90.88888889 101. ]

**Np.zero(), np.ones(), and np.eye**

matrix4 = np.zeros([3,5])

print(matrix4)

[[0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0.]

[0. 0. 0. 0. 0.]]

matrix5 = np.ones([3,5])

print(matrix5)

[[1. 1. 1. 1. 1.]

[1. 1. 1. 1. 1.]

[1. 1. 1. 1. 1.]]

Matrix6 = np.eye(5)

Print(matrix6)

array([[1., 0., 0., 0., 0.],

[0., 1., 0., 0., 0.],

[0., 0., 1., 0., 0.],

[0., 0., 0., 1., 0.],

[0., 0., 0., 0., 1.]])

**Reshape an array**

Arr4 = array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# reshaping the array arr4 to a 2 x 5 matrix

arr4\_reshaped = arr4.reshape((2,5))

arr4\_reshaped

[[0 1 2 3 4]

[5 6 7 8 9]]

**Numpy Math – it applies the log or exp to the entire array**

np.exp(2)

7.389

arr5 = np.array([2,4,6])

np.exp(arr5)

[ 7.3890561 , 54.59815003, 403.42879349]

np.log(arr5)

[0.69314718, 1.38629436, 1.79175947]

**Bringing arrays together and splitting them**

# defining two arrays

arr7 = np.arange(1,6)

print('arr7:', arr7)

arr8 = np.arange(3,8)

print('arr8:', arr8)

**Adding two arrays**

arr7: [1 2 3 4 5]

arr8: [3 4 5 6 7]

print([arr7 + arr8]) # to add arrays

[ 4, 6, 8, 10, 12]

**Matrix Math Options**

print('Addition: ',arr7 + arr8)

print('Subtraction: ',arr8 - arr7)

print('Multiplication:' , arr7 \* arr8)

print('Division:', arr7 / arr8)

print('Inverse:', 1 / arr7)

print('Powers:', arr7 \*\* arr8)

**Multiply Matrices**

matrix9 = np.arange(1,10).reshape(3,3)

print('First Matrix: \n',matrix9)

matrix10 = np.arange(11,20).reshape(3,3)

print('Second Matrix: \n',matrix10)

print('')

# taking linear algebra matrix multiplication (some may have heard this called the dot product)

print('Multiplication: \n', matrix9 @ matrix10)

First Matrix:

[[1 2 3]

[4 5 6]

[7 8 9]]

Second Matrix:

[[11 12 13]

[14 15 16]

[17 18 19]]

Multiplication:

[[ 90 96 102]

[216 231 246]

[342 366 390]]

[1, 2, 3] # dot product takes the first row and first column and multiplies them and then adds them

[11, 14, 17]

(1 \* 11) + (2 \* 14) + (3 \* 17) = 90 #row 1 and col1

(4 \* 12) + (5 \* 15) + (6 \* 18) = 231 #row 2 and col 2

**Transpose matrix**, which flips rows and columns

np.transpose(matrix9)

**Find minimum and maximum**

[[1 2 3]

[4 5 6]

[7 8 9]]

print('Minimum value: ',np.min(matrix9))

1

print('Maximum value: ',np.max(matrix9))

9

**Using np.random.rand function**

* The np.random.rand returns a random NumPy array whose element(s) are drawn randomly from the uniform distribution over (0,1). (including 0 but excluding 1).
* **Syntax** - np.random.rand(d0,d1)
  + d0,d1 – It represents the dimension of the required array given as int, where d1 is optional.

# Generating random values in an array with values between 0 and 1

rand\_mat = np.random.random(5)

print(rand\_mat)

[0.12652582 0.44350917 0.42789096 0.41140856 0.51249542]

# generates a 5x5 matrix with default values between 0 and 1

rand\_mat = np.random.rand(5,5)

print(rand\_mat)

[[0.15897709 0.29626597 0.05876492 0.2005198 0.01645951]

[0.45041024 0.7729172 0.99423285 0.05213956 0.89593565]

[0.83286412 0.66050001 0.42347676 0.06206908 0.85969301]

[0.15609053 0.41029304 0.50527114 0.2155752 0.42457115]

[0.76053592 0.02674577 0.52745842 0.40180852 0.05392995]]

**Using np.random.randn function**

* The np.random.randn returns a random numpy array whose sample(s) are drawn randomly from the standard normal distribution (Mean as 0 and standard deviation as 1)
* **Syntax** - np.random.randn(d0,d1)
  + d0,d1 – It represents the dimension of the output, where d1 is optional.

# Generating random values in an array with mean 0 and stdev 1

rand\_mat2 = np.random.randn(5)

print(rand\_mat2)

[-1.08936182 0.78956269 -1.2910328 -0.70308053 0.39640194]

#creates a 5x5 matrix with a mean of zero and a stdev of 1

rand\_mat2 = np.random.randn(5,5)

print(rand\_mat2)

[[ 1.22746832 0.67504852 -0.67885935 0.56865204 0.24147464]

[-1.22100278 0.24070193 0.10744234 -0.42801558 -0.45360308]

[ 0.84785493 -0.6212646 -0.62798733 0.87693217 -0.23308803]

[-1.28337996 0.70824889 -1.35018084 -0.76135179 0.81499775]

[ 0.04256977 -0.56186767 0.74862098 0.02948513 -0.61084276]]

**Using np.random.randint function**

* The np.random.randint returns a random numpy array whose element(s) are drawn randomly from low (inclusive) to the high (exclusive) range.
* **Syntax** - np.random.randint(low, high, size)
  + low – It represents the lowest inclusive bound of the distribution from where the sample can be drawn.
  + high – It represents the upper exclusive bound of the distribution from where the sample can be drawn.
  + size – It represents the shape of the output.

#creates array with 10 random variables between 1 and 5

rand\_mat3 = np.random.randint(1,5,10)

print(rand\_mat3)

[3 3 1 2 2 1 3 2 3 2]

# Generating random values in a matrix

rand\_mat3 = np.random.randint(1,10,[5,5])

print(rand\_mat3)

[[3 4 1 1 7]

[8 5 1 5 3]

[9 1 6 7 8]

[3 1 5 8 1]

[5 8 8 4 1]]

**Accessing elements in an array**

rand\_arr = [ 1.60534066 -1.89203407 0.79529344 -0.43814757 -0.17525549 1.14784651

**-0.12352287** 1.53604362 0.32258216 -0.49382765]

print(rand\_arr[6])

-0.12352287

#selects fifth to eighth label

print(rand\_arr[4:9])

[-0.17525549 1.14784651 -0.12352287 1.53604362 0.32258216]

#selects fourth moves three and selects, moves three more and selects

print(rand\_arr[np.arange(3,10,3)])

[-0.43814757 -0.12352287 -0.49382765]

**Logical Operators and Arrays**

rand\_arr = [ 1.60534066 -1.89203407 0.79529344 -0.43814757 -0.17525549 1.14784651

-0.12352287 1.53604362 0.32258216 -0.49382765]

rand\_arr>0

#boolean type

array([ True, False, True, False, False, True, False, True, True,

False])

#selects elements just like pandas

rand\_arr[rand\_arr>0] – to select just positive numbers

[1.60534066 0.79529344 1.14784651 1.53604362 0.32258216]

**Accessing Elements**

[[-0.4753336 -1.63251732 1.21470477 0.25034319 -0.06482419]

[ 1.9321961 -0.74053112 **-0.13865433** 1.43186983 0.29810242]

[ 0.06306887 1.62473405 -0.09906922 1.78446534 -0.10619123]

[ 1.13001826 -0.98828209 0.71641193 0.52472935 1.09577225]

[-0.25616457 -0.31299623 -0.12310303 0.86322067 -1.49812694]]

#selects row label 1

Print(rand\_mat[1])

[ 1.9321961 , -0.74053112, -0.13865433, 1.43186983, 0.29810242]

#selection row label 1 and then the second label element

print(rand\_mat[1][2])

-0.13865433

#selects row labels 0 to 1 and then columns labels 1 to 2

print(rand\_mat[0:2,1:3])

[[-1.63251732 1.21470477]

[-0.74053112 -0.13865433]]

**Change values in Array**

[ 1.60534066 -1.89203407 0.79529344 -0.43814757 -0.17525549 1.14784651

-0.12352287 1.53604362 0.32258216 -0.49382765]

# changes value of the element at row label (3 and 4) to a 5

rand\_arr[3:5] = 5

print(rand\_arr)

[ 1.60534066 -1.89203407 0.79529344 5. 5. 1.14784651

-0.12352287 1.53604362 0.32258216 -0.49382765]

# changes the elements that are greater than 0 to 65

rand\_arr[rand\_arr>0] = 65

print(rand\_arr)

[65. , 65. , 65. , 65. , 65. ,

65. , -0.12352287, 65. , 65. , -0.49382765]

**Overwriting values in matrix**

# selects rows labels 1 and 2 and selects column labels 3 and 4 and turns these four cells to 0

rand\_mat3[1:3,3:5] = 0

Matrix before modification:

[[3 4 1 1 7]

[8 5 1 5 3]

[9 1 6 7 8]

[3 1 5 8 1]

[5 8 8 4 1]]

Matrix after modification:

[[3 4 1 1 7]

[8 5 1 0 0]

[9 1 6 0 0]

[3 1 5 8 1]

[5 8 8 4 1]]

**Extracting rows and columns**

rand\_mat = array([

[-0.4753336 , -1.63251732, 1.21470477, 0.25034319, -0.06482419],

[ 1.9321961 , -0.74053112, -0.13865433, 1.43186983, 0.29810242],

[ 0.06306887, 1.62473405, -0.09906922, 1.78446534, -0.10619123],

[ 1.13001826, -0.98828209, 0.71641193, 0.52472935, 1.09577225],

[-0.25616457, -0.31299623, -0.12310303, 0.86322067, -1.49812694]])

# selects row labels 0 to 1 and then selects columns 0 to 2

sub\_mat = rand\_mat[0:2,0:3]

print(sub\_mat)

[[-0.4753336 -1.63251732 1.21470477]

[ 1.9321961 -0.74053112 -0.13865433]]

#changes all elements to 3

sub\_mat[:] = 3

print(sub\_mat)

[[3. 3. 3.]

[3. 3. 3.]]

**Change the entire matrix – this does not look correct!!!**

sub\_mat = 3 # this changes the sub mat withing the full matrix

rand\_mat

[ 3. , 3. , 3. , 0.25034319, -0.06482419],

[ 3. , 3. , 3. , 1.43186983, 0.29810242],

[ 0.06306887, 1.62473405, -0.09906922, 1.78446534, -0.10619123],

[ 1.13001826, -0.98828209, 0.71641193, 0.52472935, 1.09577225],

[-0.25616457, -0.31299623, -0.12310303, 0.86322067, -1.49812694]

**Saving and Loading NumPy Array**

np.save('/content/drive/MyDrive/Python Course/saved\_file\_name',randint\_matrix1)

loaded\_arr = np.load('/content/drive/MyDrive/Python Course/saved\_file\_name.npy')

**SciPy** [**@**](#TOC)

* Pmf = probability mass function. It creates the probability distribution for discreet variables. Use to find the probability of one or more P(X=x)
* Pdf = probability density function. It creates the probability distribution for continuous vars
* Cdf = cumulative density function. Sums the probabilities across multiple x vars
* Ppf = given the alpha, we can then calculate the value of X and can be use for both discreet and continuous

**Binomial Distribution**

# import the required function

from scipy.stats import binom

# use the binom.pmf() function to generate the probability distribution

binomial = binom.pmf(k=k, n=n, p=p)

* pmf is probability mass function and is used to calculate the probabilities for the entire array or range of values. The chart of the pmf is below
* if you need to find the probability of a single x value, you can refer to the pmf and find the probability associated with that value. If you take 1 minus the probability of the single value, that will give you and probability for not being that specific value

# declare the sample size

n = 10

# declare p which represents the probability of success, i.e., the probability that a visitor will end up buying a souvenir

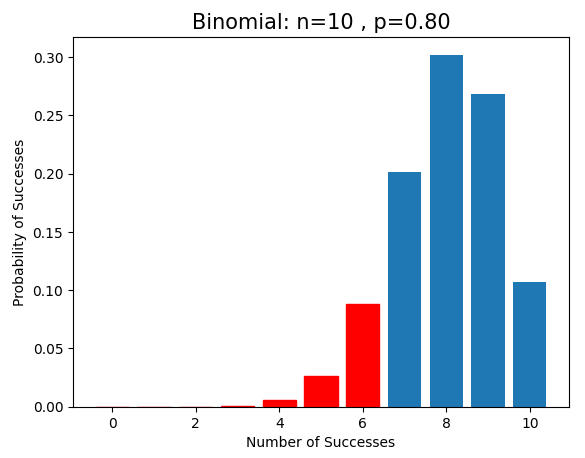
p = 0.80

# declare different possible number of visitors selected in a numpy array

k = np.arange(0,11)

print(k)

[ 0 1 2 3 4 5 6 7 8 9 10]



For binomial distributions, the cdf() function is used to find the cumulative probability for at most k successes in n trials where the probability is p.

# calculate cdf for 6 people buying something P(X<=6)

binom.cdf(k=6, n=10, p=0.8)

P(X<=7) the cdf function will sum prob 7, prob 8, prob 9 and prob 10 Answer = 0.879

**Uniform Distribution**

x = np.linspace(1, 5, 50)

probs = uniform.pdf(x, loc=1, scale=4)

uniform.cdf(x=3, loc=1, scale=4)

* cdf the cumulative density function is used because we are summing over multiple values
* answer = 0.5
* 3 is the midpoint and median

uniform.ppf(q=0.5, loc=1, scale=4)

* ppf is percentile point and allows us the reverse the question
* in the code above it is asking for the midpoint in a uniform distribution where it starts at 1 and is four units long, aka each unit has a 0.25 probability
* The answer to the 50th percentile is 3 hours

**A red and blue squares

Description automatically generated**

**Normal Distribution**

z-stat = Xbar – mu / stdev for population

z-stat = Xbar – mu / stdev/sqrt(n) for sample

P-value is the probability of the z-stat being equal to the null hypothesis

There is p-value associated with every z-stat. For example z-stat=1.96 has a p-value of .025 for a two-tailed test

**Calculate the probability that a value is below x**

Prob\_less\_than\_x = norm.cdf(z-stat)

Prob\_less\_than\_x = norm.cdf(xbar, mu, sigma) for the population

Prob\_less\_than\_x = norm.cdf(xbar, mu, sigma/sqrt(n)) for sample

This calculates the area under the normal curve to the left

**Calculate the probability that a value is above x**

Prob\_less\_than\_x = 1 - norm.cdf(z-stat)

Prob\_more\_than\_x = 1 - norm.cdf(xbar, mu, sigma) for the population

Prob\_more\_than\_x = 1 - norm.cdf(xbar, mu, sigma/sqrt(n)) for sample

When we take 1 minus, this calculates the area under the normal curve to the right

**Calculate the value associated with the x percentile**

Score\_x\_percentile = norm.ppf(.95) #confidence interval

Score\_x\_percentile = norm.ppf(percentile, mu, sigma)

Insert the percentile and this will return the x value associated with that percentile

A red and blue line graph

Description automatically generated

**Inverval estimate**

**Z-score interval estimate for pop mean when pop std dev is known**

np.round(norm.interval(0.95, loc=xbar, scale=sigma/sqrt(n),2)

* .95 is the confidence interval spread
* Loc is the sample mean
* Scale is the sample stdev
* Np.sqrt(100) the 100 is the sample size
* ,2 is a two tailed test

**T-test interval estimate for pop mean when pop std dev is unknown**

np.round(t.interval(0.95, df=99, loc=180, scale=6 /np.sqrt(100)), 2)

* .95 is the confidence interval spread
* Df is the degrees of freedom which are the sample size minus one (n-1)
* Loc is the sample mean
* Scale is the sample stdev
* Np.sqrt(100) the 100 is the sample size
* ,2 is a two tailed test

**Visualizations** [**@**](#TOC)

**Matplotlib**

#boxplot

plt.boxplot(df['Value'], vert=True, patch\_artist=True)

# Line plot

plt.plot(df['Year'], data['Sales'], marker='o', linestyle='-', linewidth=2)

# Count plot

counts = df['Category'].value\_counts()

plt.bar(counts.index, counts.values)

# Scatter plot

plt.scatter(df['X'], df['Y'], color='blue')

# Scatter plot with trendline

plt.figure(figsize=(8, 6))

plt.scatter(df['X'], df['Y'], color='blue')

# Trendline

z = np.polyfit(df['X'], df['Y'], 1)

p = np.poly1d(z)

plt.plot(df['X'], p(df['X']), color='red', linestyle='--')

#This generates histogram for all numeric variables

1. df.hist(figsize=(12,12))
   1. Plt.show()

#This is using chaining for plots

1. Df.[‘col1’].value\_counts().sort\_values(ascending=False).plot(kind='bar' ,figsize=(20,8))

# this creates a line chart by aggregating daily sales into months

df.resample('M')['sales'].sum().plot(kind='line', marker='o', figsize=(12, 6))

* Resample allows you to aggregate to a different frequency

#this groups by category and finds mean for boxplots

df.groupby('category')['value'].mean().plot(kind='box', figsize=(8, 6))

#this takes the percents and builds a bar chart

df['category'].value\_counts(normalize=True).sort\_values(ascending=False).plot( kind='bar', color='teal', figsize=(10, 6))

**Seaborn**

1. Load the visualization libraries
   1. import matplotlib.pyplot as plt
   2. import seaborn as sns
2. Customization
   1. plt.title('Title')
   2. plt.xlim(min value,max value)
   3. plt.ylim(min value, max value)
   4. plt.xlabel('xlabel')
   5. plt.ylabel('ylabel')
   6. plt.size(‘col1’) #changes the size of the dots
   7. plt.legend(loc=” “)
      1. Upper left, upper right, lower left, lower right, center, best
   8. plt.axis(“off”) removes axis
   9. plt.figure(figsize=(10,7)) to adjust size of plot
   10. plt.xticks(rotation=90)this rotates labels up and down
   11. plt.show() – is plot without any text
   12. sns.set.(style=” “) for gridlines
       1. darkgrid, whitegrid, white
3. Options within the code line
   1. Color = “” to change color
   2. Bins and binwidth for histograms
   3. kde=True or False to show a smoothed distribution
   4. hue=“” is a third dimension and is shown by creating different colors
   5. errorbar=(‘ci’, False) to turn off conf int in line charts
   6. ci=True or False to turn off conf int in line charts
   7. style=”” is often used with hue to visualize a third dimension. Hue is color while style changes the marker or line
   8. markers=True or False – adds markers to line plots
   9. col='coln' this is used to add additional charts based on the value of col. If coln has two options, you will get two scatterplots based on each option
   10. lowess = True turns on lowess line for regplots
   11. kind=”” changes the plot to the kind defined and is used with catplot, relplot, and jointplot
4. Histogram
   1. sns.histplot(data=df, x=’col1’)
   2. sns.histplot(data=df, x='price', bins=5, binwidth=10, hue=’col2’, kde=True)
      1. hue is color based on a column value
      2. kde is a smoothing curve
   3. Optimal bin math
      1. bin width = (2\*IQR)/n^1/3
      2. bins = Range/binwidth
   4. customization options
      1. plt.title('Histogram:Price')
      2. plt.xlim(3000,50000)
      3. plt.ylim(0,70)
      4. plt.xlabel('Price of cars')
      5. plt.ylabel('Frequency')
      6. sns.histplot(data=df, x='price',color='orange');
   5. Histogram Grid
      1. g = sns.FacetGrid(df, col="col1")
      2. g.map(sns.histplot, "col2");
         1. This creates a grid for each unique value in col1 in terms of col2.
         2. For example, col1 is car type and col2 is price
         3. We would get a price histogram for each car type

A graph with blue bars

Description automatically generated

1. Box Plot
   1. sns.boxplot(data=df, x='col1', color=’color’);
   2. sns.boxplot(data=df, x='col1', y='col2') ;
   3. Customization
      1. plt.title('col1')
      2. plt.xlim(30,300)
      3. plt.xlabel('col1')
      4. sns.axes\_style('whitegrid')
2. Bar Graph
   1. sns.countplot(data=df, x='col1');
   2. sns.countplot(data=df, x=’col1’, hue='col2');
   3. Customization
      1. plt.figure(figsize=(20,7))
         1. Width and height can be adjusted if the chart is jumbled
      2. plt.xticks(rotation=90) rotates labels 90 degrees
      3. plt.title('Barplot:Engine-type')
      4. plt.ylim(0,180)
      5. plt.xlabel('Engine-type')
      6. plt.legend(bbox\_to\_anchor=[1, 1]);
         1. This allows you to move the legend to upper right
3. Bar Plot
   1. This allows one to do a bar graph but also group by a second column
   2. plt.figure(figsize = (15,5))
   3. sns.barplot(data = df, x = 'Club', y = 'Total\_Points', errorbar=('ci', False))
   4. plt.xticks(rotation = 90)
   5. plt.show()
   6. may\_spending['label'].value\_counts().plot.bar();

#you can add a .plot.bar() to the end of value counts

1. Line plot
   1. sns.lineplot(data = df , x = 'col1' , y = 'col2');
      1. default includes confidence interval
   2. sns.lineplot(data = df , x = 'col1' , y = 'col2', errorbar=('ci', False));
      1. turns off ci's
   3. sns.lineplot(data=df, x = 'col1' , y = 'col2', errorbar=('ci', False) ,hue='col3');
      1. Creates multiple lines based on another categorical var
   4. sns.lineplot(data = fmri, x="timepoint", y="signal", hue="region", style="region", errorbar=('ci', False), markers = True);
      1. the hue and style columns break col2 into multiple lines and then color them differently. You can also add markers if you like

A graph with blue and orange lines

Description automatically generated

1. Scatter Plot
   1. sns.scatterplot(data=df, x='col1', y='col2');
   2. sns.scatterplot(data=df, x='col1', y='col2', hue='col3');
      1. This colors the dots by another dimension – col3
   3. sns.scatterplot(data=df, x='col1', y='col2', hue='col3', style='col3');
      1. hue provides different colors while style provides a different marker style
2. Scatter Plot with line of best fit
   1. sns.lmplot(data=df, x='col1', y='col2', ci=False);
      1. ci is the default which you can turn off
   2. sns.lmplot(data=df, x='col1', y='col2', hue='col3', ci=False);
      1. this creates a scatter plot for each col3 value and colors them differently

A graph of blue and orange dots

Description automatically generated

1. sns.lmplot(data=df, x='curb\_weight', y='horsepower', col='fuel\_type');
   * + 1. This creates multiple scatter plots based on the fuel type dimension

A graph of different types of data

Description automatically generated with medium confidence

1. sns.regplot(data=df, x="horsepower", y="price", lowess=True)
   * + 1. Lowess plot
2. Violin Plot
   1. sns.violinplot(data=df, x='col1'); #use y for vertical plot of same data
      1. provides both distribution, IQR, median, and whiskers
   2. sns.violinplot(data=df, x='col1', y='col2', orient='v');
      1. This is a col1 plot, split by the col2 dimension. Orient v is up and down
3. Split chart is multiple scatter plots
   1. sns.stripplot(data=df, x='col1', y='col2', jitter=True);
      1. creates a scatter like plot for col1 and split by col2 dimension
      2. jitter keeps dots from stacking on top of each other
4. Pairplot
   1. sns.pairplot(data=df[['normalized\_losses','wheel\_base','curb\_weight','engine\_size','price','peak\_rpm']])
      1. This provides scatter and histograms for all combinations of columns
   2. sns.pairplot(data=df, vars=['wheel\_base', 'curb\_weight', 'engine\_size', 'price'], hue='number\_of\_doors');
      1. this creates different colors
   3. sns.pairplot(data=df, vars=['wheel\_base', 'curb\_weight', 'engine\_size', 'price'], corner=True);
      1. This just provides the lower scatter plots
      2. \*\*\*\*\*This is the preferred option\*\*\*\*\*
5. Heatmap
   1. sns.heatmap(data=df[['wheel\_base','curb\_weight','engine\_size','price']].corr());
      1. Creates correlation heat map
   2. sns.heatmap(data=df[['wheel\_base','curb\_weight','engine\_size','price']].corr(), annot=True, fmt='0.2f', cbar=False);
      1. cannot add corr coefficients and cbar=False removes the legend
   3. sns.heatmap(data=df[['wheel\_base','curb\_weight','engine\_size','price']].corr(), annot=True, cmap='YlGnBu');
      1. cmap allows you to adjust the color. This is yellow green and blue heatmap

**ROC Curve**

yhat1 = model1.predict(X\_test) # predict probabilities

yhat1 = yhat1[:, 0] # keep probabilities for the positive outcome only

fpr, tpr, thresholds1 = roc\_curve(y\_test, yhat1) # calculate roc curves

gmeans1 = np.sqrt(tpr \* (1-fpr)) # calculate the g-mean for each threshold

ix = np.argmax(gmeans1) # locate the index of the largest g-mean

print('Best Threshold=%f, G-Mean=%.3f' % (thresholds1[ix], gmeans1[ix]))

# plot the roc curve for the model

pyplot.plot([0,1], [0,1], linestyle='--', label='No Skill') #diagonal line

pyplot.plot(fpr, tpr, marker='.') #plots the ROC curve

pyplot.scatter(fpr[ix], tpr[ix], marker='o', color='black', label='Best')

# axis labels

pyplot.xlabel('False Positive Rate')

pyplot.ylabel('True Positive Rate')

pyplot.legend()

# show the plot

pyplot.show()

**MatPlotLib**

**Visualization Loops and Combinations**

# Select independent variables (excluding index)

var\_list = df.columns.tolist() # Since 'date' is index, it's not included

# Create a grid of subplots (2 columns × enough rows for all variables)

num\_vars = len(var\_list)

rows = (num\_vars // 2) + (num\_vars % 2) # Ensures enough rows for all variables

fig, axes = plt.subplots(nrows=rows, ncols=2, figsize=(15, 5 \* rows)) # Dynamically sized

# Flatten axes array for easier iteration

axes = axes.flatten()

# Loop through each variable and create scatter plots

for i, col in enumerate(var\_list):

ax = axes[i]

ax.scatter(df[col], df['grocery\_sales'], color='green', alpha=0.6)

ax.set\_xlabel(col, fontsize=10)

ax.set\_ylabel('Grocery Sales', fontsize=10)

ax.set\_title(f'Scatter Plot: {col} vs Grocery Sales')

# Hide empty subplots if `var\_list` has an odd number of elements

for j in range(i + 1, len(axes)):

fig.delaxes(axes[j])

# Adjust layout and show all plots

plt.tight\_layout()

plt.show()

**Data Pre-Processing** [**@**](#TOC)

**Mount Drive**

* from google.colab import drive
* drive.mount(‘/MIT\_Class’)

**Convert Series or data objects into a df**

* pd\_df = pd.DataFrame(df.data object)

**Describe the df**

* df.shape #will give the number of rows and columns
* df.info() #to see data type and non-null value counts
* df.isnull().sum() #gives you the count of missing values in each column
* data.isnull().sum()/data.isnull().count() \* 100 #provides the percent
* df.nunique() #give you the number of unique values in each column
* df[‘col1’].unique() #give you the unique values in a column
* df.describe().T to see count, mean, min, max, and quartiles
* df.describe(include=['object']) will show count, unique, top, and freq

**Counting values**

* df[‘col1’].value\_counts()
* Data["Target"].value\_counts()/Data["Target"].count()
* df['col1'].apply(type).value\_counts() #to see the count of value types in a column
* cat\_cols = data.select\_dtypes(include='object').columns.tolist() #this selects all object or numeric columns
* df.groupby(['col1'])['col2'].agg(col1 and col2\_label='unique',col1 and col2 label='count')
  + This groups by col1
  + Col2 is the column that will be the basis of the agg
  + Agg unique provides the unique combination of values with Col1 and Col2
  + Agg count counts the unique combination of values within col1 and col2
* for column in cat\_cols:
* print(data[column].value\_counts())
* print("-" \* 50) # this put a line ----- between column counts
* for column in cat\_cols:
* print(data[column].value\_counts(normalize=True))#percents
* print("-" \* 50)

**Formats**

* pd.set\_option('display.float\_format', lambda x: '%.3f' % x) to keep float decimals to 3 places
* df[“date”] = pd.to\_datetime(df[“date”], format = ‘%d-%m-%Y’) to convert to dates
* df[“col1”] = df[“col1”].astype(‘int’) to convert to integer, float, category or string

**Clean the data**

* data.duplicated().sum() to get a count of duplicate rows
* df = df.drop\_duplicates(inplace=True) this removes rows that are the same
* df['col1'] = data['col1'].replace(['missing','inf'],np.nan) this replaces the missing or inf values witn NaN
* df.drop(columns=[‘col1’, ‘col2’])
* df.drop(2328, inplace = True) #drop a row based on index value

**Clean messy columns with both numeric and text**

* df['col1'] = df['col1'].apply(lambda x: float(x.split(' ')[0]) if pd.notnull(x) else np.nan)
  + for columns that have numbers and text together but we only want the numeric data, e.g., ‘42 mpg’
  + split(‘ ‘) splits the values in col1 where there is a space and keeps the first column [0], and float converts to float
  + if the cell is not null, it runs the split function else if it is null, in inserts a NaN

**Fill in missing values - Columns to impute**

* reqd\_col\_for\_impute = ["Education\_Level", "Marital\_Status", "Income\_Category"]
* # Simple Imputer which replaces missing values with the Mode
* imputer = SimpleImputer(strategy="most\_frequent")

# this transforms the columns in place

* data1[reqd\_col\_for\_impute] = imputer.fit\_transform(data1[reqd\_col\_for\_impute])

# Example of using mode to impute

* imputer\_mode = SimpleImputer(strategy="most\_frequent")
* X\_train[["Enrolled\_university","Education\_level","Major\_discipline","Experience","Company\_type","Last\_new\_job"]] = imputer\_mode.fit\_transform(
* X\_train[["Enrolled\_university","Education\_level","Major\_discipline","Experience","Company\_type","Last\_new\_job"]])
* X\_test[["Enrolled\_university","Education\_level","Major\_discipline","Experience","Company\_type","Last\_new\_job"]] = imputer\_mode.transform(
* X\_test[["Enrolled\_university","Education\_level","Major\_discipline","Experience","Company\_type","Last\_new\_job"]])

**Outlier definition, calculation, table, and fix**

* #create the num\_col list.
* num\_cols = data.select\_dtypes(include='number').columns.tolist()
* # to find the 25th percentile and 75th percentile for the numerical columns. It is like a loop but it is not. The quantile fucntion is applied to each column in the list
* Q1 = data[numeric\_columns].quantile(0.25)
* Q3 = data[numeric\_columns].quantile(0.75)
* IQR = Q3 - Q1  Inter Quantile Range (75th percentile - 25th percentile) is also calculated for each column in list
* lower\_whisker = Q1 - 1.5\*IQR    #Finding lower and upper bounds for all values. All values outside these bounds are outliers
* upper\_whisker = Q3 + 1.5\*IQR

#creates a table of the vars created above

iqr\_df = pd.DataFrame({

    'Column': Q1.index,  # Column names

    'Q1 (25th percentile)': Q1.values,  # Q1 values

    'Q3 (75th percentile)': Q3.values,  # Q3 values

    'IQR': IQR.values,  # Interquartile range

    'Lower Whisker': lower\_whisker.values,  # Lower whisker

    'Upper Whisker': upper\_whisker.values  # Upper whisker})

#This provides the percent of values above or below the whiskers

((data.select\_dtypes(include=["number"]) < lower\_whisker) | (data.select\_dtypes(include=["number"]) > upper\_whisker)).sum() / len(data) \* 100

# This fixes the outliers

def treat\_outliers(df, col):

    """

    treats outliers in a variable

    col: str, name of the numerical variable

    df: dataframe

    col: name of the column

    """

    Q1 = df[col].quantile(0.25)  # 25th quantile

    Q3 = df[col].quantile(0.75)  # 75th quantile

    IQR = Q3 - Q1                # Inter Quantile Range (75th perentile - 25th percentile)

    lower\_whisker = Q1 - 1.5 \* IQR

    upper\_whisker = Q3 + 1.5 \* IQR

    # all the values smaller than lower\_whisker will be assigned the value of lower\_whisker

    # all the values greater than upper\_whisker will be assigned the value of upper\_whisker

    # the assignment will be done by using the clip function of NumPy

    df[col] = np.clip(df[col], lower\_whisker, upper\_whisker)

    return df

**Set the index**

* df = df.set\_index(‘col1’) sets col1 as index
* df=df.reset\_index return back to autonumber
* df. Reset\_index().set\_index(“col1”) changes the index from the current index to col1

**Identifying odd data and replacing**

* odd\_values = pd.DataFrame(df.col1.str.isdigit()) this looks for string values in a digit formatted column
* df[odd\_values[‘col1’] == False] this prints the original df rows with the odd values
* df[‘col1’] = df[‘col1’].replace(‘?’, np.nan) this replaces a ? with NaN
* df[odd\_values[‘col1’] == False] run this again to confirm the change

**Replacing missing values**

* df['col1'] = df['col1'].fillna(data['col1'].median()) this replaced the missing values with the median

**Create Standardized Variables**

scaler = StandardScaler()

* This imports the library to scale data

data\_scaled = pd.DataFrame(scaler.fit\_transform(df), columns = df.columns)

* pd.DataFrame turns the output in to a df

scaler.fit\_tranformation(df)

* scales the data in df. It finds the mean and the stdev and then applies it to the df

columns=df.columns

* keeps the column names

**Rescale positive and negative values into a 0 to 1 range**

df['scaled\_value'] = (df['value'] - df['value'].min()) / (df['value'].max() - df['value'].min())

* rescales the values by subtracting the minimum value and dividing by the range, which is the maximum value minus the minimum value

**Create Lagged Variables**

df['lagged\_value'] = df['value'].shift(1)

* Shift function moves data down n number of places
* The shifted data at the beginning are replaced with NaN

**Create Logged Variables**

df['logged\_value'] = np.log(df['value'])

* This applies the natural log to the defined column
* All values must be positive

**Create Dummy Variables for Cross Sectional Data**

df = pd.get\_dummies(df, columns = df.select\_dtypes(include = ["object", "category"]).columns.tolist(), drop\_first = True,)

* This runs get\_dummy function.
* It creates a list of columns that are either object or category
* The drop\_first drops the first dummy column and keeps the remaining. Thus if you created a dummies for months, January would be dropped and would be the baseline for the dummy var

**Create Dummy Variables for Time Series Data**

df['dummy'] = 0

* Initialize to 0

df.loc[(df['date'] >= 'start\_date') & (df['date'] <= 'end\_date'), 'dummy'] = 1

* After start date and before end date is a 1
* All other dates have been initialized to 0

**Encoding Labels for Categorical Columns**

# this transforms variable from category to numbers

from sklearn.preprocessing import LabelEncoder

labelencoder\_RE = LabelEncoder()

X\_train['Relevent\_experience']= labelencoder\_RE.fit\_transform(X\_train['Relevent\_experience'])

X\_test['Relevent\_experience']= labelencoder\_RE.transform(X\_test['Relevent\_experience'])

**Splitting the dataset into train and test datasets**

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size = 0.2, shuffle = True, random\_state = 1)

* Splitting the data into x and y df
* 20% going into test df
* Shuffle is to randomize
* Random state means we can reproduce and will get the same test and train if we run in the future

**Checking the shape of the train and test data**

print("Shape of Training set : ", x\_train.shape)

print("Shape of test set : ", x\_test.shape)

**Filtering Columns and Rows [@](#_top)**

**COLUMNS**

**1) Create a list to filter columns, process data, and create a new column**

#Define the list of columns

columns\_to\_sum = ['A', 'B', 'C']

#the column list in the input into the sum function

df['sum\_columns'] = df[columns\_to\_sum].sum(axis=1)

In this simple example, we define a list, link it to the df, and then bolt on .sum(axis=1), to create a new column in df. The axis=1 tells python to sum horizontally.

**2) Use the list and ability to sum across columns to calculate department mix**

#Define the list

depts = ['Dairy','Frozen Foods','Grocery','Hbc','Liquor And Cigars','Natural Choice', 'Non Foods','Bakery','Deli', 'Meat And Seafood','Produce','Wall Deli']

#sum across the columns to create the total file

dept['total'] = dept[depts].sum(axis=1)

#loop through each department spend and divide it by the total department spend

for col in num\_cols:

dept[f'{col}\_mix'] = dept[col]/dept['total']

**3) Use a lambda function to sum across columns**

# Define the columns for each group

non\_loyal = ['GO', 'LP', 'PO', 'UN']

loyal = ['PR', 'VL']

all\_columns = ['GO', 'LP', 'PO', 'UN', 'PR', 'VL'] # Rename to avoid confusion

# Use the apply wrapper and row as the function variable to process one row at a time

# row[non\_loyal].sum() filters the df columns by those defined in the list and then they are summed. The same for row[all\_columns].sum()

# this allows us to calculate the mix of loyal and non\_loyal spend

shabits['non\_loyal\_mix'] = shabits.apply(lambda row: row[non\_loyal].sum() / row[all\_columns].sum(), axis=1)

shabits['loyal\_mix'] = shabits.apply(lambda row: row[loyal].sum() / row[all\_columns].sum(), axis=1)

**4) Using the assign method**

#.assign() adds new columns to the DataFrame and is great for chaining operations. This allows to combine multiple lambda functions in one space

# Calculate the total spend and department mix in one line using .assign()

dept = dept.assign(total=dept[depts].sum(axis=1),

Dairy\_mix=lambda x: x['Dairy'] / x['total'],

Grocery\_mix=lambda x: x['Grocery'] / x['total'] )

**5) Use transform to apply a function and then broadcast the result back to the original df**

#Transform() is useful when you want to apply a function to groups of data and then broadcast the results back to the original df. It allows you to perform operations without changing the shape of the df. Tranform returns a series with the same index as the original df

a) For calculating department mix, first calculate the denominator – Total Sales

b) loop through departments

for dept in departments:

df[f'{dept}\_Mix'] = df[dept] / df['Total\_Sales'].transform(lambda x: x)

* The transform(lambda x: x) part takes the Total\_Sales column, and since it's already calculated, it simply returns the same values but they are the same across stores.
* The division operation calculates the mix for each department by dividing its sales by the total sales for that row.
* The resulting DataFrame has new columns showing the mix for each department.

Store Sales Expenses Total\_Sales\_Per\_Store

0 A 100 50 250

1 A 150 60 250

2 B 200 70 450

3 B 250 80 450

4 C 300 90 300

See how Store A total sales is replicated or duplicated across both A rows. Thus it maintains the same number of rows

**6) Use the Lambda function to find the standard score for a column**

# use the apply to process one column or row at a time. In this case, this code is processing an one entire column at a time so the parameter is column and not row. Scipy.stats computes the z-scores for all values in a given column at once.

df[num\_cols] = df[num\_cols].apply(lambda col: zscore(col, nan\_policy='omit'))

**7) Using the lambda function and col parameters to normalize an entire column**

# Apply Min-Max scaling using lambda function

# this is applied to an entire column, using the min, max, and x value to create the normalized value

df\_normalized = df.apply(lambda col: (col - col.min()) / (col.max() - col.min()))

**8) Using lambda to recode a numeric column into categories**

# Categorize values in column 'A' as 'High' or 'Low' based on a threshold

# this function takes input from df[“A”] and runs it through the conditional statement to create a new column df[“A Category”]

df['A\_Category'] = df['A'].apply(lambda x: 'High' if x > 25 else 'Low')

**9) Using lambda to apply a mathematical function to a column**

# Square the values in columns 'A' and 'B'

# this function takes in inputs from col A and B and then squares them

df\_squared = df[['A', 'B']].apply(lambda col: col \*\* 2)

**10) Using lambda and a condition to make changes to a column**

# Apply different transformations to columns based on their names

# If the column is 'A', the values are doubled; otherwise, they are halved.

df\_transformed = df.apply(lambda col: col \* 2 if col.name == 'A' else col / 2)

**11) Use lambda to calculate the NPS**

# Step 1: Count the number of promoters (9-10) and detractors (1-6)

promoters\_count = df['A'].apply(lambda x: 1 if x in [9, 10] else 0).sum()

detractors\_count = df['A'].apply(lambda x: 1 if x >= 1 and x <= 6 else 0).sum()

# Step 2: Calculate the total number of responses

total\_responses = len(df)

# Step 3: Calculate the proportions

promoters\_proportion = promoters\_count / total\_responses

detractors\_proportion = detractors\_count / total\_responses

# Step 4: Calculate the NPS score

nps\_score = promoters\_proportion - detractors\_proportion

**FeatureTools** [**@**](#TOC)

Setup Feature Tools with a transaction, customer, and product table

* Customer table is customer id and sign up date
* Transaction table is transaction id, customer id, product id, transaction date, price paid, quantity, and sales
* Products table is product id, category, and price
* Cutoff\_time is customer id and cutoff\_time. This is where we draw the line for the analysis. No data will be considered past this date for each customer

**Three tables used to build an entity set**

1) Create an Entity Set and use Featuretools to generate features for each unique customer and each unique cutoff\_time

**# Creating an entity set for Featuretools. The entity set is like defining a relational db. The entities are the tables but we also include or define the relationships between tables and define the key cutoff times**

es = ft.EntitySet(id="retail")

Adding entities to the entity set

**#customer master**

es = es.entity\_from\_dataframe(

entity\_id="customers",

dataframe=customers,

index="customer\_id",

time\_index="signup\_date"

)

**#transaction table**

es = es.entity\_from\_dataframe(

entity\_id="transactions",

dataframe=transactions,

index="transaction\_id",

time\_index="transaction\_time"

)

**#product masters**

es = es.entity\_from\_dataframe(

entity\_id="products",

dataframe=products,

index="product\_id"

)

**# Defining relationships**

relationship1 = ft.Relationship(es["customers"]["customer\_id"], es["transactions"]["customer\_id"])

relationship2 = ft.Relationship(es["products"]["product\_id"], es["transactions"]["product\_id"])

es = es.add\_relationship(relationship1)

es = es.add\_relationship(relationship2)

**# Specifying features**

features, feature\_defs = ft.dfs(

entityset=es,

target\_entity="customers",

agg\_primitives=["mean", "sum", "count"],

trans\_primitives=["month", "weekday"],

cutoff\_time=cutoff\_time,

cutoff\_time\_in\_index=True,

verbose=True

)

Specifying features

* ft.dfs this creates the feature matrix
* entityset=es is the entity set name
* target\_entity=”customers” this is the group by column
* agg\_primitives are what math will be applied to the groupby
* trans\_primitives are how time is defined in this example. Weekdays get translated into 0-6 and months get translated into 0-11. It creates representation of dates that can be applied to ML
* cutoff\_time = cutoff\_time is how cutoff\_time is connected
* cutoff\_time\_in\_index=True adds the cutoff\_time to the df
* verbose=True is just what type of information is output

**One table broken into three to create an entity set**

Create the individual tables from the one large table

* customers = data[['customer\_id', 'customer\_name', 'signup\_date']].drop\_duplicates()
* products = data[['product\_id', 'product\_name', 'category', 'price']].drop\_duplicates()
* transactions = data[['transaction\_id', 'customer\_id', 'product\_id', 'transaction\_time', 'purchase\_amount']]

**Repeat the steps above**

**PCA** [**@**](#TOC)

Principal components is the rotation of the coordinate system to find new axis that capture the maximum variance in the data. These principal components are also known as the eigenvectors which represent our new stretched or transformed coordinate system. The length of the eigenvector is the eigenvalue. The larger the eigenvalue, the more variance explained.

“Learning the transformation” means calculating these eigenvectors (PCs) and their associated eigenvalues, which define how the data should be stretched or transformed to capture the maximum variance in the data.

* Fit in **fit\_**transformed figures out the PCs

“Projecting Onto Principal Component Axes” is the process of transforming each data point from the original dataset onto the new coordinate system. Projection refers to the process of mapping each original data point onto the new axes.

* Transformed in fit\_**transformed** is tranforming the original dataset onto the new coordinate system

For each PC there is a coefficient for each feature. This is the **PC loadings or components**. They represent the impact of that feature on that PC. The higher number means more impact and +/- is the relationship with the PC. These values are stored in **pca.components\_**

The PC loading are represented by a matric with PC as columns and the features or variables as rows. The values are the loadings and they usually range from -1 to 1

The PC loadings, which are the coefficients, times the observation inputs, give us the PC scores, which are then used in regression, clustering, etc.

PC1(obs 1) = coef1\*X1 + coef2\*X2 + …..coefn\*Xn

This is done for each observation to translate the original data points and coordinate system into the new stretched PC coordinate system. This new PC coordinate system maximizes the variance explained and also creates orthogonal new PC or features.

**1) Scale the data**

scaler=StandardScaler() # this is just the library that allows you to standardize vars. It contains the isntructions and methods needed to standardize the data

data\_scaled=pd.DataFrame(scaler.fit\_transform(data), columns=data.columns)

* scaler in the instructions and methods to stardardize the vars
* fit\_transform(input df) standardizes the data. The fit is computing the mean and standard deviation for each feature (column or variable). The transform is applying the mean and stdev to the original df and computing the stardardized vars
* Column=data.columns is all columns in the input df
* Pd.DataFrame converts the standardized scores into a df

**2) PCA analysis**

**n**=data\_scaled.shape[1]

**pca** = PCA(n\_components=**n**, random\_state=1)

pca\_scores = pd.DataFrame(**pca**.fit\_transform(data\_scaled))

exp\_var = **pca**.explained\_variance\_ratio\_

#Defining the number of principal components to generate

n=data\_scaled.shape[1]

* This returns the number of columns in the data\_scaled df
* n = 1 means we will generate the same number of PC as original vars

#Finding principal components for the data

pca = PCA(n\_components=n, random\_state=1)

* This code brings in the instructions and methods to do the PCA. ChatGPT
* It specifies the number of PC to compute. Because our n=1, we will generate the same number of PC as vars
* random\_state=1 is the seed so we can reproduce the results

#create new dataset with PC as the columns and all of the rows. The values are the pca scores

pca\_scores = pd.DataFrame(pca.fit\_transform(data\_scaled))

* pca brings in the instructions and methods from the line of code above
* fit\_transform – fits the PCA model to the standardized data and then transforms the data into the principal components
* data\_scaled is the input data
* The output is an array where each row is a transformed sample, and each column corresponds to a PC. These are the pca scores

#The percentage of variance explained by each principal component

exp\_var = pca.explained\_variance\_ratio\_

* pca brings in the instructions and methods from the line of code above
* To the right of =, that give use the total variance explained by each PC
* The output is a 1D array where each element corresponds to the percent of variation explained by eac PC in descending order
* Exp\_var is a var that stores the explained variance for each PC. Like a list [0.52, 0.30, 0.12, 0.03]

**3) Chart the cumulative explained variance**

# visualize the explained variance by individual components

plt.figure(figsize = (10,10)) # sets the size of the chart

sns.lineplot(x=range(1, 8), y=exp\_var.cumsum(), marker='o', linestyle='--')

* x=range(1,8) creates a range from 1 to 7 which is the number of PCs
* exp\_var.cumsum() – exp\_var has the PC variance explained values and cumsum() is a function that sums of the inputs

plt.title("Explained Variances by Components")

plt.xlabel("Number of Components")

plt.ylabel("Cumulative Explained Variance")

**4) Create the Factor or PC loads. The output is a df**

pca\_loads = ['PC1','PC2','PC3']

* create a list for the index labels

data\_pca = pd.DataFrame(np.round(pca.components\_[:3,:],2),index=pc\_comps,columns=data\_scaled.columns)

* pca brings in the instructions for calculating the loadings
* components\_ stores the coeficients or weights or loadings for each PC and feature. Each feature receives a weight for each PC. The bigger the value, the more important the feature is for that PC. Can also be +/- depending on the relationship
* [:3,:],2 slices the data to get the first three rows or PCs and the : grabs all of the columns. The 2 is for rounding
* Index=pc\_comps sets the index of the df to the list of pc\_comps which is the list PC1, PC2, and PC3
* columns=data\_scaled.columns – this grabs the column names of the stardardized df and inserts them into the new df with the new PC scores

data\_pca.T

* flips the data frame so we can view the PCs as columns and features as rows

**5) highlight values**

def color\_high(val): # defines function and the input is the high or low values

    if val <= -0.40: # you can decide any value as per your understanding. If value is LT -.4 it is pink

        return 'background: pink'

    elif val >= 0.40:

        return 'background: skyblue'

data\_pca.T.style.applymap(color\_high)

* data\_pca.T refers to the transposed df

.style.applymap(color\_high)

* .style. applies styling to a dataframe
* .applymap is used to activate the function
* .applymap(color\_high) applies the function to each cell of the df individually
  + Applymap() passes each cell value in the df to color\_high function. The return value from color\_high determines the styling for that cell

**6) Plotting PC1 and PC2**

df\_concat = pd.concat([data\_pca1, data], axis=1)

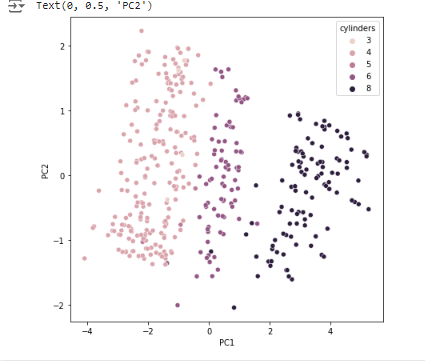
* This combines the original df with the PC1 data to link up the cylinders which are used in the hue statement

plt.figure(figsize = (7,7))

sns.scatterplot(x = 0, y=1, data = df\_concat, hue='cylinders')

plt.xlabel("PC1")

plt.ylabel("PC2")



**Clustering** [**@**](#TOC)

**Libraries**

# To scale the data using z-score

from sklearn.preprocessing import StandardScaler

# Importing clustering algorithms

from sklearn.cluster import KMeans

from sklearn.mixture import GaussianMixture

from sklearn\_extra.cluster import KMedoids

from sklearn.cluster import AgglomerativeClustering

from sklearn.cluster import DBSCAN

# Silhouette score

from sklearn.metrics import silhouette\_score

import warnings

warnings.filterwarnings("ignore")

**Scale the data**

scaler = StandardScaler()

* This imports the library to scale data

data\_scaled = pd.DataFrame(scaler.fit\_transform(df), columns = df.columns)

* pd.DataFrame turns the output in to a df
* scaler.fit\_tranformation(df) – scales the data in df. It finds the mean and the stdev and then applies it to the df
* columns=df.columns – keeps the column names

**K-Means**

# Empty dictionary to store the SSE for each value of K

sse = {}

* this initialized a dictionary that will store the SSE for each cluster solution

# Iterate for a range of Ks and fit the scaled data to the algorithm.

# Use inertia attribute from the clustering object and store the inertia value for that K

for k in range(1, 10):

* will create cluster solutions for 1 to 9 clusters

    kmeans = KMeans(n\_clusters = k, random\_state = 1,n\_init="auto").fit(df\_scaled)

* create a k means clustering model with k clusters. Random\_state is a seed for reproducability
* the n\_clusters = k pulls the number from the loop
* fit\_scaled(df\_scaled) – fits the Kmeans model to the scaled data (df\_scaled)

    sse[k] = kmeans.inertia\_

* This measures how well the clusters fit the data. It stores the calculated SSE (aka inertia) for the current k solution in the sse dictionary with k as the key

**Elbow Plot**

sns.lineplot(x=list(sse.keys()), y=list(sse.values()))

* For this lineplot we do not have a data= option because we are passing in the values directly as a list

plt.xlabel("Number of cluster")

plt.ylabel("SSE")

plt.show()

**Silhouette Scores**

Start from 2 because silhouette requires at least two cluster option to calculate the score

Inertia measures the sum of squared distances between each data point and the centroid. It is know as within cluster sum of squares (WCSS). It is the sum of the euclidean distances.

The disadvantage of inertia is that it does not account for how well-separated the clusters are

The silhouette score looks at both within cluster distance and between cluster distance, so it is more robust

# Empty dictionary to store the Silhouette score for each value of K

sc = {}

# Iterate for a range of Ks and fit the scaled data to the algorithm. Store the Silhouette score for that K

for k in range(2, 10):

    kmeans = KMeans(n\_clusters = k, random\_state = 1).fit(df\_scaled)

* Like above this takes the loop values and creates a cluster solution for each

    labels = kmeans.predict(df\_scaled)

* This assigns each data point to a cluster

    sc[k] = silhouette\_score(df\_scaled, labels)

* We swap out the SSE from above with the silhouette score which results in a more definitive elbow plot
* SSE just measures distance between the point and the cluster center – inter cluster distance – the lower the score the better
* Silhouette measures both how close the points are to the cluster center AND how far the points of one cluster are from the points in other clusters – both inter and intra cluster distances – the higher the score the better

# Elbow plot

sns.lineplot(x=list(sc.keys()), y=list(sc.values()))

plt.xlabel("Number of cluster")

plt.ylabel("Silhouette Score")

plt.show()

# with the optimal number of clusters identified, build the final model

kmeans = KMeans(n\_clusters = 3, random\_state = 1).fit(df\_scaled)

* This just runs the final model and then fits it to the scaled data

#create cluster labels

cluster\_labels = kmeans.predict(df\_scaled)

* This creates the numpy vector which contains a cluster label or segment value for each row

# With the model built and cluster labels defined, add predicted labels to the original data and the scaled data

#scaled data

df\_scaled['KMeans\_Labels'] = cluster\_labels

#original data

df['KMeans\_Labels'] = cluster\_labels

* This adds the predicted cluster values to the data df with the column name ‘Kmeans\_Labels’
* We don’t need to merge or concat because the rows in the df and vector align. This is the direct way to merge two pieces of data. Just need to make sure the rows correspond to each other as a perfect one-to-one join

#count the size of the clusters

data['KMeans\_Labels'].value\_counts()

# Calculating the mean and the median of the original data for each label to understand the different segment profiles

mean = data[num\_cols].groupby('KMeans\_Labels').mean()

* This finds the means for each num\_col for each cluster

median = data[num\_cols].groupby('KMeans\_Labels').median()

* This finds the median for each num\_col for each cluster

df\_kmeans = pd.concat([mean, median], axis = 0)

* Merges the mean and median df. When you groupby and have a function like mean or median, that output is a df so it can be concated

df\_kmeans.index = ['group\_0 Mean', 'group\_1 Mean', 'group\_2 Mean', 'group\_0 Median', 'group\_1 Median', 'group\_2 Median']

* This changes the row index labels from 0, 1, 2… to the list above. This will then be transformed in the code below and they will become the column names

df\_kmeans.T

**K-Medoids Clustering**

# run medoid clusters

kmedo = KMedoids(n\_clusters = 3, random\_state = 1)

* Brings in the instructions for building the kMedoid clusters.
* Defined to be 3 and including the seed for reproducability

kmedo.fit(data\_scaled)

* This finds the optimal cluster solution and fits the data to it

data\_scaled\_copy['kmedoLabels'] = kmedo.predict(data\_scaled)

* This builds the cluster vector

data['kmedoLabels'] = kmedo.predict(data\_scaled)

* This merges the cluster vector to the original data

#count cluster sizes

#run mean and median code again (see above)

# run the boxplot and cross tab visualizations again

**Gausian Mixture Model**

# run the model

gmm = GaussianMixture(n\_components = 3, random\_state = 1)

* Bring in the instructions

gmm.fit(data\_scaled)

* Build the segments and fit the data

data\_scaled\_copy['GmmLabels'] = gmm.predict(data\_scaled)

* Create the segment vector

data['GmmLabels'] = gmm.predict(data\_scaled)

* Merge the segment vector back to the df

#count cluster sizes

#run mean and median code again (see above)

# run the boxplot and cross tab visualizations again

**Hierarchical Clustering**

#Load library

from scipy.cluster.hierarchy import dendrogram, linkage

# The List of all linkage methods to check

methods = ['single',  'average',  'complete']

* These are the the three methods we will loop through

# Create a subplot image before defining subgroups

fig, axs = plt.subplots(len(methods), 1, figsize = (20, 15))

* This define the figure and the axes.
* Plt.subplots() creates a figure and a set of subplots within that figure
* It returns two objects
  + Fig this is the figure object and contains all of the subplots. It is the container for the entire figure
  + Axs this is an array of subplot axes within the figure. Each axes is an individual plot
* Len calculates the number of elements in the list methods, which is three. This defines the number of subplots to create
* The 1 specifies the number of colums in subplot, so there will be one column and three subplots stacked on top of each other

# Enumerate through the list of all methods above, get linkage and plot dendrogram

for i, method in enumerate(methods):

    Z = linkage(data\_scaled, metric = 'euclidean', method = method)

    dendrogram(Z, ax = axs[i]);

    axs[i].set\_title(f'Dendrogram ({method.capitalize()} Linkage)')

    axs[i].set\_ylabel('Distance')

plt.figure(figsize = (20, 7))

plt.title("Dendrograms")

dend = dendrogram(linkage(data\_scaled, method = 'complete'))

plt.axhline(y = 9, color = 'r', linestyle = '--')

# Clustering with 4 clusters

hierarchical = AgglomerativeClustering(n\_clusters = 4, affinity = 'euclidean', linkage = 'complete')

hierarchical.fit(data\_scaled)

data\_scaled\_copy['HCLabels'] = hierarchical.labels\_

data['HCLabels'] = hierarchical.labels\_

#count cluster sizes

#run mean and median code again (see above)

# run the boxplot and cross tab visualizations again

**DBSCAN**

# run model

dbs = DBSCAN(eps = 1)

* Load instructions

data\_scaled\_copy['DBSLabels'] = dbs.fit\_predict(data\_scaled)

* Fit the model

data['DBSLabels'] = dbs.fit\_predict(data\_scaled)

* Merge segments onto original data

#count cluster sizes

#run mean and median code again (see above)

# run the boxplot and cross tab visualizations again

**Regression** [**@**](#TOC)

**Building Models**

x\_train = x\_train.astype(float)

y\_train = y\_train.astype(float)

* Convert all vars in model to float

import statsmodels.api as sm

* Import library

x\_train = sm.add\_constant(x\_train)

* Statsmodel API does not add a constant by default

x\_test = sm.add\_constant(x\_test)

* Adds constant to test data

model = sm.OLS(y\_train, x\_train).fit()

* Calls the OLS function from statmodel
* y is the dependent and x is the features
* fit() applies the OLS instructions to the df
* fit() also creates the coefficients
* fit() produces the results table which includes the coef, pvalues, rsq, Durbin Watson,
* oldmodel1 is the fitted model object and from this object you can call the coef, summary stats, and other data to make predictions on new data

model.summary()

print(model.summary())

* Retrieve and print the model summary

**Three function to evaluate the model – RMSE, MAE, Rsq, Adj Rsq and MAPE**

**Function parameters**

* Predictors are the independent var actual data. The source is the df from which the model was run
* Targets are the dependent var actual values that we are trying to predict. The source is the df from which the model was run
* Predictions are the target predicted values generated by the model. The source is the .predict() method on the training df
* Model is the regression model object that has been previously trained (fitted) using the predictor vars (predictors) and the dependent var (target)
  + The model outputs are the predicted values, which are stored in Predictions above. Thus Predictions are an output of model.
  + These outputs are called in the code as model.predict(predictors)
  + Model is an argument in the model\_performance\_regression function because it allows us to compare methods/models such as decision tree regression or neural networks.

# 1) Function to compute adjusted R-squared

def adj\_r2\_score(predictors, targets, predictions):

r2 = r2\_score(targets, predictions)#r2\_score calculation

    n = predictors.shape[0] #counts the observations from df

    k = predictors.shape[1] #counts the vars in the model from df

    return 1 - ((1 - r2) \* (n - 1) / (n - k - 1)) #adj Rsq formula

# 2) Function to compute MAPE

def mape\_score(targets, predictions): #inputs are actual and pred y

    return np.mean(np.abs(targets - predictions) / targets) \* 100

* np.abs(targets – predictions) #calculates the abs value of errors
* mean() # finds the mean of the abs values

# 3) Function to compute different metrics to check performance of a regression model

def model\_performance\_regression(model, predictors, target):

    """

Function to compute different metrics to check regression model performance

    model: regressor

    predictors: independent variables

    target: dependent variable

    """

pred = model.predict(predictors) # Predicted values which are inputs into the measures below. Each of the measures below use scikit functions which come before the () while the values in the () are the parameters or inputs

r2 = r2\_score(target, pred) # Compute R-squared

    adjr2 = adj\_r2\_score(predictors, target, pred)#Compute adj R-sq

    rmse = np.sqrt(mean\_squared\_error(target, pred)) #Compute RMSE

    mae = mean\_absolute\_error(target, pred)#Compute MAE

    mape = mape\_score(target, pred)# Compute MAPE

    # Creating a dataframe of metrics

    df\_perf = pd.DataFrame(

        {

            "RMSE": rmse,

            "MAE": mae,

            "R-squared": r2,

            "Adj. R-squared": adjr2,

            "MAPE": mape,

        },

        index=[0],

    )

    return df\_perf

lin\_reg\_test = model\_performance\_regression(model, x\_test, y\_test)

lin\_reg\_test

* This function uses the trained model (model) to generate predictions based on the test data (x\_test)
* This calls the function above and then prints it out
* model – this is the trained regression model
* X\_test – this is the text independent vars
* Y\_test – this is the dependent var from the test data and it is used to calculate Rsq, Adj Rsq, RMSE, MAE, and MAPE

**Quick summary for training and test data performance**

print("Performance on train data: ")

model\_performance\_regression(model, x\_train, y\_train)

print("Performance on test data: ")

model\_performance\_regression(model, x\_test, y\_test)

**Dropping Insignificant Vars**

#manually drops insig vars from both test and train

x\_train2 = x\_train.drop(['Insurance\_Yes','staff\_available','Visitors with Patient'], axis = 1)

x\_test2 = x\_test.drop(['Insurance\_Yes','staff\_available','Visitors with Patient'], axis = 1)

# Train the model

model2 = sm.OLS(y\_train, x\_train2).fit()

# Get the model summary

model2.summary()

# Run model\_performance\_regression to evaluate the test data performance

**Diagnostics**

1. Multicollinearity
2. Residual mean = 0
3. Normal dist of error terms
4. Linear vars
5. Heteroscedacticity

**VIF**

from statsmodels.stats.outliers\_influence import variance\_inflation\_factor

def checking\_vif(train):

    vif = pd.DataFrame()#creates the vif df

    vif["feature"] = train.columns #creates a list of vars from train data

    # Calculating VIF for each feature

    vif["VIF"] = [

variance\_inflation\_factor(train.values, i)

for i in range(len(train.columns))

    ]

    return vif

* variance\_inflation\_factor is the vif function
* train is the independent vars and is defined by train.columns and is stored in the vif df as feature
* i is the counter and loops through each item in the var list
* The i range is defined as the number of columns in the var list

**Residual mean = 0**

residual = olsmodel2.resid #resid is an attribute which is where residuals are stored. This statement creates are residual series named residual

df[‘residuals’] = residual #merge residuals to the df

residuals.mean() #gives the residual mean

**Residual Distribution**

sns.histplot(residual, kde=True)

**Linear Variables**

# Predicted values

fitted = olsmodel2.fittedvalues

# Plotting Residuals VS Fitted Values

sns.residplot(x = fitted, y = residual, color="lightblue")

plt.xlabel("Fitted Values")

plt.ylabel("Residual")

plt.title("Residual PLOT")

plt.show()

* Plot residuals and fitted values in a scatter plot. The results should show no pattern
* Loess can be used to detect any patterns

**Heteroscedacticity**

import statsmodels.stats.api as sms

from statsmodels.compat import lzip

name = ["F statistic", "p-value"]

test = sms.het\_goldfeldquandt(residual, x\_train2)

lzip(name, test)

* sms.het\_goldfeldquandt is the function
* Inputs are the model residuals and independent vars (predictors)
* The Goldfeld-Quandt test checks for heteroscedasticity by dividing the dataset into two subsets and comparing the variances of the residuals between these subsets. It is particularly sensitive to changes in variance related to the values of the predictors.
* Lzip(name, test) This function returns a tuple containing the F statistic, the p-value, and an alternative measure of variance.

**Cross Validation**

* Need to use sklearn for validation because statsmodel does not offer this
* Need to rerun model

# Fitting linear model

from sklearn.linear\_model import LinearRegression

from sklearn.model\_selection import cross\_val\_score

linearregression = LinearRegression() #loads the lin regression inputs

cv\_Score11 = cross\_val\_score(linearregression, train\_features\_scaled\_new6, train\_target\_log, cv = 10)

* Train\_features\_scaled\_new6 is the independent variable df which was defined above
* Train\_target is the dependent df which was also defined above
* Cv is the number of folds
* The output is an array stored in cv\_Score11. It holds each of the Rsq for each of the 10 folds

cv\_Score12 = cross\_val\_score(linearregression, train\_features\_scaled\_new6, train\_target\_log, cv = 10, scoring = 'neg\_mean\_squared\_error')

* Same as above except this version captures the MSE
* Thus we have the Rsq and MSE to calculate the folds and then compare to the training dataset

print("RSquared: %0.3f (+/- %0.3f)" % (cv\_Score11.mean(), cv\_Score11.std()\*2))

* Rsq mean

print("Mean Squared Error: %0.3f (+/- %0.3f)" % (-1\*cv\_Score12.mean(), cv\_Score12.std()\*2))

* MSE mean

Output is the Rsq and MSE mean for all 10 folds

**Creating forward forecast with lagged variables**

**Classification** [**@**](#TOC)

**Type 1 Error – False positive**

**Type 2 Error – False negative**

A diagram of a graph

Description automatically generated with medium confidence

Precision – How often a classifier is right given the positive predicted outcome

* Predicted accuracy = true positives/(true positives + false positives)
* What proportion of positively predicted outcomes did we get right
* High precision is low false positive and high false negatives

Recall or sensitivity – How much the classifier is right based on the actual outcomes

* Actual accuracy = true positives/(true positives + false negatives)
* What proportion of actual outcomes did we get right
* High recall is low false negative and high false positives

There is a tradeoff between these two

A white background with black text

Description automatically generated

* High precision often result in lower recall and vice versa
* Being sensitive to predicted accuracy (precision) likely means we flag a lot of positive outcomes, so we will catch most fraud but this will also result in low precision and many false positives
* Being sensitive to the actual outcomes data (recall) will lead to low false positives but will also result in missing some fraud cases
* So both measures are important and need to be balanced
* This is similar to over and under fitting for models
* F1 score is the score that combines both precision and recall and is what we try to maximize. The F1 score is the harmonic mean of X and Y

A math equation with black text

Description automatically generated

Accuracy is another measure to evaluate the model, but only works well if the counts in the confusion matrix are fairly balanced.

* Total correct/all outcomes
* TN+TP/TN+FN+TP+FP

In an imbalanced example, like fraud where 99% of cases are not fraud, building a model that is 99% accurate is not difficult

With imbalanced datasets, Recall is preferred

* Recall focuses on the accuracy in the actual positive column
* Recall = TP/(TP+FN)
* The recall percent includes false negatives, which are those outcomes we predicted true but were actually false.
* With an imbalanced dataset, we want to keep false negatives low because positive counts will be very high

Precision becomes more relevant when the cost of a false positive is high and the cost of a false negative is very low.

* Precision focuses on the accuracy in the predicted positive column
* Precision = TP/(TP+FP)
* Precision employs false positives, which are those we predicted negative but were actually positive
* FP is included because their cost is high so we want to minimize FP

A diagram of a graph

Description automatically generated with medium confidence

**Binomial Distribution**

Build the distribution based on the control group probabililty. Then compare the target group probability to see if there is a meaninful difference. The distribution shape is based on the probability of a positive outcome. The mean is the actual number of positive outcomes.

A graph of a normal distribution

Description automatically generated

**Decision Trees**

* DT’s are nonlinear classifiers
* Order matters. If the first split is on var a and another tree’s first split is on var b, you will have different results.
* Greedy algorithm picks the variables that explain the most variation first, aka minimize impurity. Purity and Entropy are inversely related. The higher the purity, the lower the entropy and vice versa.
* Entropy is a measure of uncertainty. How clear is the pattern? If it is 50/50, risk is high if 90/10, entropy is low. We can see from the chart below, at 0/100 and 100/0, entropy is 0, while 50/50 has an entropy score of 1, which is where the derivative is 0, aka it is the max

A white background with black text

Description automatically generated

* Information Gain – if a feature is introduced and reduced entropy, it will result in information gain. Our goal is to minimize entropy and maximize information gain.
* The greedy algorithm selects features in the order of the entropy score. Smaller scores selected first
* Can also use the Gini Index to replace Entropy. If favors larger groups



* Adding features usually reduces misclassification on the training set but we see misclassification increasing on the test dataset. Bias vs training trade-off.
* Pruning can help to balance bias and variance by reducing over fitting. Pruning removes branches that are not very important which may reduce the training fit somewhat but makes a larger improvement on the test dataset
* Pruning is inefficient because you first classify all of the features and then it goes back and removes those that are less important

**Ensemble Learning**

* We can reduce variance if we average a number of independent random variables
* Averaging techniques reduce variance
* Bagging is Ensemble Learning and combines bootstraping + aggregation
  + Bootstrap – sample data with replacement over and over. The example was a dataset with 1000 observations and each sample with replacement was 200. We then created 20 different bootstrap samples and 20 different trees.
    - Use the holdouts or unsampled data to validate the model
  + Aggregate all of the models and average the results or voting
    - If our 20 trees classify 15 as 0 and 5 as 1, voting would say 0 is the estimated outcome

**Random Forest**

* We are losing interpretability but improving the accuracy of the prediction. We are approaching the black box but our misclassification is less
* Different from bagging above, Random forest random samples both the observation and the features with replacement. Bagging only randomly samples observations and not features
* This increases independence of decision tree which increases the prediction power
* Hyperparameters – we want trees that are independent of each other. One way to make them more independent, randomly sample only a few features for each tree. This increases independence but increases missclassification so there is a tradeoff

A diagram of a tradeoff

Description automatically generated

* The trick is to balance the number of features in a tree with the correlation between trees. More features, the more correlations and more correlation increases error. But more features also reduces error within each tree but the correlation between trees increases error across all trees

**Classification and Regression Tree (CART)**

* Very similar to above but we have continuous data so we can use regression
* Can use Random Forest with CART

**Overfitting**

* Single tree – prune weaker branches
* Multiple trees
  + Bagging which takes multiple samples and then votes
  + Random Forest which takes multiple samples of observations and features and then votes

**Python Code for Classification**

**Prepare the data**

**# Creating list of dummy columns**

dummy = ['BusinessTravel', 'Department','Education', 'EducationField','EnvironmentSatisfaction', 'Gender',  'JobInvolvement','JobLevel', 'JobRole', 'MaritalStatus' ]

# Creating dummy variables

df = pd.get\_dummies(df, columns = dummy, drop\_first = True)

# Mapping overtime and attrition

dict\_OverTime = {'Yes': 1, 'No':0}

dict\_attrition = {'Yes': 1, 'No': 0}

# Overwrites the data in the df with the map function

df['OverTime'] = df.OverTime.map(dict\_OverTime)

df['Attrition'] = df.Attrition.map(dict\_attrition)

# Create the dependent and independent features

Y= df.Attrition

* Just the index and the Attrition values in the df

X= df.drop(columns = ['Attrition'])

* All columns except Attrition

**Split the data**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size = 0.3, random\_state = 1, stratify = Y)

* X\_train, X\_test, y\_train, y\_test – these are the four dfs that will be created
* **train\_test\_split()** is a function that splits up the df or array
* X is the feature set or input data
* Y is the dependent variable
* Test\_size is the percent of observations that are in the test df
* Random\_state = 1 ensures reproducability
* Stratify = Y ensures the split is stratified based on the target variable. This is important when dealing with imbalanced classes

**Standardize the data for logistic and SVM**

# Scaling the data

scaler=StandardScaler()

# Fit\_transform on train data

df\_scaled\_X\_train=pd.DataFrame(scaler.fit\_transform(X\_train), columns=X\_train.columns)

# Transform on test data

df\_scaled\_X\_test=pd.DataFrame(scaler.fit\_transform(X\_test), columns=X\_test.columns)

**#Evaluation Metrics**

def metrics\_score(actual, predicted):

    print(classification\_report(actual, predicted))

    cm = confusion\_matrix(actual, predicted)

plt.figure(figsize=(8,5))

sns.heatmap(cm, annot=True,  fmt='.2f', xticklabels=['Not Attrite', 'Attrite'], yticklabels=['Not Attrite', 'Attrite'])

    plt.ylabel('Actual')

    plt.xlabel('Predicted')

    plt.show()

* classification\_report is a function reports out the detailed classification metrics
* confusion matrix is a function that computes the confustion matrix and a summary of the prediction results

**Logistic Regression**

**Build the model is statsmodel for Challenger Example**

**Statsmodel syntax**

model = SM.logit(formula='Y ~ X1 + X2', df=data)

result = model.fit()

* fits the log reg model to the data. Result will hold the fitted model objects – estimated coef, p-values, std error, etc

**sklearn Logistic syntax**

* lg = LogisticRegression()
* lg.fit(X\_train,y\_train)

**Results from the Challenger model**

A screenshot of a computer

Description automatically generated

* The model shows -0.1466 temp coefficient. This is interpreted as a drop in temperature by one degree will increase the chance of an O ring failure by 15%
* The overall probability that an O ring would fail at 36 degrees
  + P(Y=1(O ring failure) | X=36 degrees = e^(B0 + Bx)/(1 + e^(B0 + Bx))
  + e^(7.4 – 0.1466\*36)/(1 + e^(7.4 – 0.1466\*36)) =0.89 which is the prob for one ring to fail
  + Since ring failure is independent of the other rings, the probability of all five failing would be 0.89^5 or 57%

**Build the model using sklearn for the employee churn model**

# Fitting logistic regression model

lg\_model = LogisticRegression()#loads log reg

lg\_model.fit(df\_scaled\_X\_train, y\_train)#runs model on training data

train\_pred = lg\_model.predict(df\_scaled\_X\_train)

* applies training model to train data

test\_pred = lg\_model.predict(df\_scaled\_X\_test)

* applies training model to the test data

# Checking the performance on the training data

metrics\_score(y\_train, train\_pred)

* this calls the function above. It provides us with a confusion matrix and various performance metrics

# Checking the performance on the test dataset

metrics\_score(y\_test, test\_pred)

* same except it calls the test estimates and the actual test data

**Printing the coefficients of logistic regression**

# Finding the odds.

lg\_coef\_ = lg\_model.coef\_[0] #this give the coefficients which should be used to sort the features. The coefficients are not interpretable and need to be converted to odds to make sense

lg\_odds\_ = np.exp(lg\_model.coef\_[0]) # this give the odds which are easier to interpret. For the odds, a one unit increase in the dependent results in the odds of the event increasing or decreasing by x%

# Creating two dataframes, one for coef and one for odds

lg\_coef = pd.DataFrame(lg\_coef\_, X\_train.columns, columns = ['coef'])

lg\_odds = pd.DataFrame(lg\_odds\_, X\_train.columns, columns = ['odds'])

# Merge the dfs and then sort

lg\_coef\_odds = pd.merge(lg\_coef, lg\_odds, left\_index=True, right\_index=True).drop('features\_y', axis=1).rename(columns={'features\_x' : 'features'})

lg\_coef\_odds = lg\_coef\_odds.sort\_values(by='coef', ascending=False)

#chart the coef

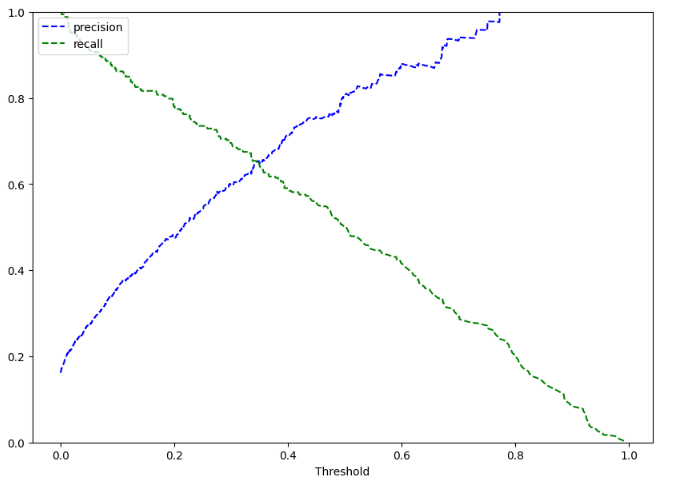
plt.figure(figsize=(7,15))

sns.barplot(data=lg\_coef\_odds, x='coef', y='features');

**Create the Precision/Recall Curve**

This chart is useful because the default threshold is if the prob is GT 50%, then it is a 1 and LT 50% it is a 0.

This chart shows us if we should adjust the 50% to something less. Our optimal threshold is 30%, thus, if it is GT 30% it is a one and LT 30% is a zero.

****

# Predict\_proba gives the probability of each observation belonging to each class

y\_scores\_lg\_model=lg\_model.predict\_proba(df\_scaled\_X\_train)

precisions\_lg\_model, recalls\_lg\_model, thresholds\_lg\_model = precision\_recall\_curve(y\_train, y\_scores\_lg\_model[:,1])

# Plot values of precisions, recalls, and thresholds

plt.figure(figsize=(10,7))

plt.plot(thresholds\_lg\_model, precisions\_lg\_model[:-1], 'b--', label='precision')

plt.plot(thresholds\_lg\_model, recalls\_lg\_model[:-1], 'g--', label = 'recall')

plt.xlabel('Threshold')

plt.legend(loc='upper left')

plt.ylim([0,1])

plt.show()

**y\_scores\_lg\_model=lg\_model.predict\_proba(df\_scaled\_X\_train)**

* predict\_proba returns an array with two columns. Each row is an observation and each column is the probability of being in class 0 or class 1. They will equal 1 when summed. The function is applied to the dataset used to build the classification model
* lg\_model and the df (df\_scaled\_X\_train) work in conjuction to calculate the probabilities in the predict\_proba array. The coefficients come from lg\_model and the data from the df

**precisions\_lg\_model, recalls\_lg\_model, thresholds\_lg\_model = precision\_recall\_curve(y\_train, y\_scores\_lg\_model[:,1])**

* precision\_recall\_curve is the function that calculates precision, recall and the threshold values. The precision curve includes a threshold value and precision value. Threshold is the x and precision value is the y. This also holds for recall.
* The threshold value is the value that determines if calculated probability is a 0 or a 1.
* The precision\_recall\_curve inputs are y\_train, y\_score\_lg\_model[:,1]
  + Y\_train is the actual 0 and 1 label
  + y\_score\_lg\_model[:,1] is an array of probabilities for 1 occuring. [:,1] selects the probabilities corresponding to 1 from the output of the predict\_proba array.
* The precision\_recall\_curve outputs are the different precision and recall values for the different thresholds. Precision and Recall are the y values while the threshold is the x value
* Where the two lines cross balances recall and precision and is the optimal threshold which will maximize both the accuracy of the training and test models

**plt.plot(thresholds\_lg\_model, precisions\_lg\_model[:-1], 'b--', label='precision')**

**plt.plot(thresholds\_lg\_model, recalls\_lg\_model[:-1], 'g--', label = 'recall')**

* [:-1] included because there is one extra datapoint for recall and precision that is not included in threshold. Thus, this last datapoint is removed.

**Recheck the performance of both train and test models using the optimal threshhold**

#test model at optimarl threshhold of 0.35 which is where the precision and recall lines cross

optimal\_threshold=.35

train\_pred = lg\_model.predict\_proba(df\_scaled\_X\_train)

metrics\_score(y\_train, train\_pred[:,1]>optimal\_threshold)

# do the same thing for the test data

optimal\_threshold=.35

test\_pred = lg\_model.predict\_proba(df\_scaled\_X\_test)

metrics\_score(y\_test, test\_pred[:,1]>optimal\_threshold)

**Support Vector Machines**

SVM uses kernal functions. Two of the more common kf’s are Linear Kernal and RBF Kernal

# Fitting the Linear SVM option

svm = SVC(kernel='linear') # Linear kernal or linear decision boundary

model = svm.fit(X= df\_scaled\_X\_train, y = y\_train)

* Kernal = ‘linear’ tells SVM to find the best linear (straight line or hyperplane) boundary that separates the classes. These models are generally more simple and easier to interpret
* svm.fit(X=X\_train\_scaled, y=y\_train) trains the SVM model using the df provided. X\_train\_scaled is the input data and y\_train is actual 0 and 1 values.

# Run the svm model

y\_pred\_train\_svm = model.predict(df\_scaled\_X\_train)

* makes predictions given the input data
* output is an array of predicted class labels (0 or1) for each obs in the dataset.

# Run the metrics score function

metrics\_score(y\_train, y\_pred\_train\_svm)

* function defined above

**Decision Tree**

# Building decision tree model

model\_dt = DecisionTreeClassifier(class\_weight = {0: 0.17, 1: 0.83}, random\_state = 1)

* Loads the model and establishes what will be 0’s and 1’s
* 1’s get much more weight because there are fewer in the dataset

# Fitting decision tree model

model\_dt.fit(X\_train, y\_train)

* This runs the model and fits it to the training data

# Checking performance on the training dataset

y\_pred\_train\_dt = model\_dt.predict(X\_train)

* This saves the predicted values in y\_pred\_train\_dt

# run metric function

metrics\_score(y\_train, y\_pred\_train\_dt)

# Checking performance on the test dataset

y\_test\_dt = model\_dt.predict(X\_test)

* Runs the model and fits it to the test data

# run metrics on test data

metrics\_score(y\_test, y\_test\_dt)

# Plot the feature importance

importance = model\_dt.feature\_importances\_

columns = X\_train.columns # replace df with df X\_train

importance\_df = pd.DataFrame(importance, index = columns, columns = ['Importance']).sort\_values(by = 'Importance', ascending = False)

plt.figure(figsize = (13, 13))

sns.barplot(data = importance\_df, x = importance\_df.Importance, y = importance\_df.index)

* **importance = model\_dt.feature\_importances\_** returns a coefficient array. These are not really coefficients but similar. Called importance and not coefficient
* **columns = X.columns** aligns the column names with the coefficient array
* **importance\_df** creates a df which transposed the array and sorts the values

#plot the tree

features = list(X.columns)

plt.figure(figsize = (30, 20))

tree.plot\_tree(model\_dt, max\_depth = 4, feature\_names = features, filled = True, fontsize = 12, node\_ids = True, class\_names = True)

plt.show()

**Random Forest**

# Fitting the Random Forest classifier on the training data

model\_rf = RandomForestClassifier(class\_weight = {0: 0.17, 1: 0.83}, random\_state = 1)

model\_rf.fit(X\_train, y\_train)

# Checking performance on the training data

y\_pred\_train\_rf = model\_rf.predict(X\_train)

metrics\_score(y\_train, y\_pred\_train\_rf)

# Checking performance on the testing data

y\_pred\_test\_rf = model\_rf.predict(X\_test)

metrics\_score(y\_test, y\_pred\_test\_rf)

# Plot the feature importance

importance = model\_rf.feature\_importances\_

columns = X.columns

importance\_rf = pd.DataFrame(importance, index = columns, columns = ['Importance']).sort\_values(by = 'Importance', ascending = False)

plt.figure(figsize = (13, 13))

sns.barplot(data = importance\_rf, x = importance\_rf.Importance, y = importance\_rf.index)

**Hyperparameters**

Grid Search allows you to test dozens of models and rank them based on your critieria. Could be overall accuracy, recall, or precision. In the example, below, we are optimizing on recall because we want to minimize false negatives.

# Choose the type of classifier

rf\_estimator\_tuned = RandomForestClassifier(class\_weight = {0: 0.17, 1: 0.83}, random\_state = 1)

# Grid of parameters to choose from

params\_rf = {

        "n\_estimators": [100, 250],

        "min\_samples\_leaf": np.arange(1, 4, 1),

        "max\_features": [0.7, 0.9, 'auto'],

        'max\_depth': np.arange(3, 5),

}

# Type of scoring used to compare parameter combinations - recall score for class 1

scorer = metrics.make\_scorer(recall\_score, pos\_label = 1)

# Run the grid search

grid\_obj = GridSearchCV(rf\_estimator\_tuned, params\_rf, scoring = scorer, cv = 5)

grid\_obj = grid\_obj.fit(X\_train, y\_train)

# Set the classifier to the best combination of parameters

rf\_estimator\_tuned = grid\_obj.best\_estimator\_

**Deep Learning and Neural Networks** [**@**](#_top)

A NN is a combination of three mathematical functions – linear, activation, and loss functions.

Each unit/node of a Neural Network is a simple linear classifier. It is the bias and activation function that allows NN to handle non linear data.

We can create hierarchies or groups for the NN to process the data. These hierarchies help make complex mapping simpler to process

* Input > edges > simple parts > parts > objects > scenes

**A feed forward neural network has three layers**

1. input layer is passive and does no processing and only holds the data. The inputs are a vector. If you have a nxn matrix, you stack them to create a vector with nxn elements
2. hidden layers – takes the input, multiplies by a weight and adds bias. The weights are all randomly initialized
3. The outputs are also a vector and align with the problem. If it is a binary classification, then there is only one output. If there are 10 groups which you are classifying, then there are 10 elements in the vector.

**Neurons are a computational node. They receive and send inputs and outputs**

1. multiply input layer values by a weight and then sum all of the inputs
2. Compare the summed value to the threshhold
3. If it is larger than the threshold, the data will pass to the next node

**Training Of A Neural Network**

**A white background with black text

Description automatically generated**

**We need to decide several things**

1. Initialization for starting weights – go with standard methods
2. Architecture – start with an architecture that someone has already created for this type of data and then start tweaking that
   1. How many layers
   2. How many units in a layer
   3. Activation function type – Sigmoid, Than, Relu
3. Training
   1. Learning Rate or step size (start small and we have to tune ourselves)
   2. Optimization
      1. Momentum – provides a smoother path to a minumum. It updates weights by using the current weight and a fraction of the previous weight, like a weighted average
      2. Adaptive – Adapts the learning rate for each parameter based on how frequently that parameter is updated during training. Parameters that have large gradients get smaller updates and vice versa
      3. Adam – Combines both. It uses momentum to accelerate learning and adapts the learning rate for each parameter individually
4. Overfitting
   1. Early Stopping
   2. L1 and L2 Regularization
   3. Dropout

**Step 1: Forward Propagation**

**Step A:**

In forward propagation, the data points LaTeX: x_1,\:x_2,\:...,\:x_N from the input layer are propagated to a single neuron where each input is multiplied with its respective weights and then summed together. Each neuron has also an error term called bias. The sum of the bias term and the linear combination of inputs and weights is the input to the single neuron as shown in the below image.

**The weights and bias are random numbers generated within a small range, typically 0-1**

A diagram of a mathematical equation

Description automatically generated

**Step B:**

In this step, we apply a nonlinear function to this linear combination. The functions we apply to these linear combinations are also known as Activation Functions. Activation Functions are a gateway for the neuron and decides whether to move the signal forward or not. Activation Functions also introduce nonlinearity into our Neural Network.

A diagram of a mathematical equation

Description automatically generated

In the image above, we have applied a **sigmoid function** which is one of the activation functions.

**Step 2: Calculate the Loss Function or the error**

After getting the output as a result from forward propagation, we will calculate the loss using the loss function. The weights and biases are updated in such a way that the loss function is minimized.

The loss function determines the **magnitude of the error** for a given set of predictions, while backpropagation calculates how the error propagates through the network.

* Loss = (Y-Yhat)\*\*2 - squared error

The larger the loss or error, the larger the gradients that are back propogated through the network to make bigger adjustments to the weights

**Step 3: Backpropagation**

Backpropagation calculates how much each weight in the network contributes to the overall error (defined by the loss function) and updates the weights to minimize that error. The way to do it is, we take the derivative of the cost with respect to a particular weight and then we shift the value of the weights in that direction.

LaTeX: w\:=\:w\:-\:\frac{dc}{dw}

Where C is the error term and w is the weight we want to modify. These derivatives are the gradients that tell the model how much a change in each weight will affect the loss or error. Bigger error result in bigger changes to the weights.

Backpropagation relies on the chain rule of calculus to compute the gradients efficiently. The chain rule allows the error from the loss function to be propagated backward layer by layer, adjusting each weight based on how much it contributes to the error.

Backpropagation is used to update the weights and biases are known as Optimizers - Gradient Descent, SGD, Batch SGD, etc.

**Step 4: Repeat Forward and Backward Propagation until the cost function is minimized.**

We repeat Forward Propagation and Backward Propagation until the cost/objective function is minimized.

The below graphic representation shows a single iteration of forward and backward propagation. In forward propagation, first, calculate the value for each node using the input layer and the activation functions. Secondly, make the predictions using the output layer and calculate the error/loss function using the predicted and the actual labels. In backward propagation, the weights and biases are updated using derivatives to optimize the loss function.

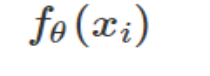
A diagram of a algorithm

Description automatically generated

**Cost function and loss function**

The cost function and loss function are synonymous (some people also call it the error function). A cost function is a measure of error between what your model predicts and what the actual value is.

The below function of theta represents the output of a model over xi variables



Then the cost function can be represented as

A black and blue text

Description automatically generated with medium confidence

Where yi is the actual output value and the difference between the actual value and the predicted value is squared and summed over all n points.

The loss function (or error) is for a single training example, while the cost function is over the entire training set (or mini-batch for mini-batch gradient descent).

**Activation Functions**

Activation functions are introduced to learn complex patterns in the data. The activation function decides whether a nueron should be activated. It takes input from previous layers and converts it to some form of input for the next layers.

The most important feature of activation functions is to introduce non-linearity into a neural network because if there is no non-linearity it's just a linear algorithm or we can say the model just tries to fit a straight line but there are more complex patterns in data that cannot be identified by a linear algorithm.

We can use different types of activation functions as given below:

**The Sigmoid Function:** It is one of the most widely used non-linear activation functions. Sigmoid transforms the values into a range between 0 and 1. It can be interpreted as the probability of a particular class. The mathematical expression for sigmoid: LaTeX: f\left(z\right)\:=\frac{\:1}{\left(1+e^{-z}\right)}

A graph of a function

Description automatically generated

A graph on a white sheet

Description automatically generated

A graph on a graph

Description automatically generated

**Bias**

The bias is part of every node or unit. It is part of the activation function and can be used to adjust the threshold, depending on the model objectives. You can make it easier or more difficult to meet the threshold and pass the data forward.

For each of the three activation functions, the bias moves the line to the left or right. If you move the bias to the right, it makes it more difficult to meet the threshold and vice verse. The shift is a parallel shift. So if the Relu above started to move in the postive direction at x=2, the bias would be 2

**The Softmax Function**

The Softmax function returns the probability of each class. Here's the equation for the Softmax activation function:

LaTeX: softmax\left(z_i\right)=\frac{exp\left(z_i\right)}{\sum_j\:exp\left(z_j\right)}

Here, the Z represents the values from the neurons of the output layer. The exponential acts as the non-linear function. Later these values are divided by the sum of exponential values in order to normalize and convert them into probabilities.

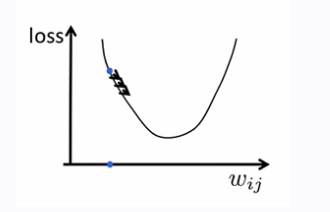
**Gradient Descent**

Gradient Descent - This is an optimization algorithm that helps us find the parameters of a machine learning model. Gradient descent uses the given equation to formulate the theta for which the error is minimum. It keeps updating the theta until convergence is achieved.

A white background with black text

Description automatically generated

Where *J(LaTeX: \theta0,LaTeX: \theta1)* is the error function for the regression. i.e.LaTeX: \sum (Y-Yhat)2



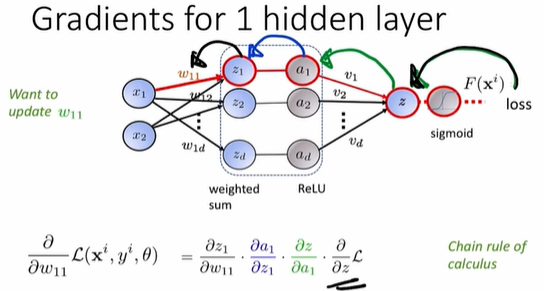
The partial derivatives with respect to the weight indicates how much the loss will change, so they help the gradient descent model determine the size of the changes to the weights.

Wnew = Wold – n \* dL/dw

n = learning rate

dL/dw is the gradient (partial derivative) of the loss function with respect to the weight. The partial derivative gives us the contribution of each weight to the error. Thus, the partial derivative tells us how to adjust each weight to minimize the loss. The chain rule allow us to compute how each weight affects the loss function, taking into account all intermediate transformations for both activations and layers

Once the weights and their contribution to the error is determined, the weights are updated using gradient descent





Each arrow is a derivative. This is the chain of derivatives.

* F(xi)-yi is just the error and is the last derivative dL/dz. F(x1) is the prediction and y1 is the actual value or desired value
* Vi is the weight v1 and is represented by the green arrow and dz/da1
* Blue term is 1 if active and 0 if inactive based on activation function. If it is zero, the the weight w11 does not matter

A diagram of a network

Description automatically generated with medium confidence

When doing multiple layers, we can reuse many of the computations. The values are memorized which speeds up processing.

**Initialization**

For large NN, very large weights can become unstable and not converge. Aka exploding gradient. Weights too small can result in a very time consuming back propagation. Aka vanishing gradient.

As such initialization is important so our weights are not to big or small

A graph of a function

Description automatically generated

Where do we initialize? It is done randomly and a normal Gaussian distribution with a mean 0. What is the variance? This impacts exploding and vanishing gradients. So we want the variance to not be too big or too small. It is common to use 2/#units

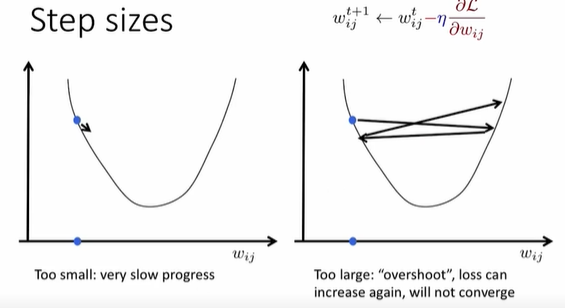
A graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of a graph of

Description automatically generated

Red line has a larger variation and converges faster. For the second chart with 30 layers, the bigger variation converges much faster than the blue which does not learn at all so it is just a flat line.

**Step Size**

The amount by which the weights are updated during the training process is referred to as the step size. Small step size means we move slowly down the gradient. Big steps can over shoot.



A graph of sizes and numbers

Description automatically generated

* For n=2.5 it is just bouncing back and forth and the cost is not falling with each epoch
* For n=0.25 it converges the fastest and is the optimal value
* For n=.025 is too small and converges slowly but is better than 2.5.

**Optimizers**

Adaptive methods such as AdaGrad

* If gradient/slope/derivative is very large or steep, we should move slowly
* If gradient/slope/derivative is small or not steep, we can move more aggressively

Momentum

* As we move into a valley, SGD will zig zag into the valley. Momentum uses averages to go more straight and converge more quickly

Adam – combine both adaptive and momentum into one method

**Overfitting and Regularization for NN**

A graph and a chart

Description automatically generated with medium confidence

**Early Stopping**

The chart on the left is measuring loss for each epoch and the chart of the right is the accuracy. The y-axis is loss.

After 300 epochs, loss does appear to ber falling, but our accuracy is not. So we could end the training process at 300 epochs, which is called an Early Stop.

Plot the accuracy on the hold out data and it will help you define when to stop.

**Other regularization to prevent overfitting**

* Like Lasso regresssion, we can add the squared norm/ weight decay which keeps weights from getting too large and overfitting. We just add a regularization term to our partial derivative.

A math equation with a plus and a positive symbol

Description automatically generated

A white background with black text

Description automatically generated

* Data augmentation – This increases your data using small pertubations that don’t change the meaning of the data, eg., rotate images. You can also average data points to help smooth out decision functions
* Drop out – changes training procedure by switching off some of the units on one pass and switching off other on the next pass. This is an ensemble method that is similar to Random Forest.
  + Note: when using this method you also have to scale the wieghts by the same percent of units that you are turning off.
* Batch Normalization which standardizes the data for the input layer. Changes the architecture by introducing another layer. By scaling the data, BN helps to keep the model from reacting too much

**CODING**

# This is used with Keras and TensorFlow and helps to manage memory usage. When creating multiple models, the session can accumulate objects and require high memory usage and lead to OOM errors. This should be called after each model. This also holds for loops

**NN Process - Train, Observe, Modify**

**1)BASE NN MODEL**

backend.clear\_session()

#set seed to results are reproducable

np.random.seed(1)

import random

random.seed(1)

tf.random.set\_seed(1)

# Initializing the ANN

model = Sequential() #layers will be added sequentially, one after another

# The amount of nodes (dimensions) in hidden layer should be the average of input and output layers, in this case 64.

# This adds the input(by specifying input dimension)

model.add(Dense(activation = 'relu', input\_dim = 9, units=64))

* Dense() a dense fully connected layer is being added to the model. This means every neuron in this layer is connected to every neuron on the previous layer or input layer
* Activation=’relu’ using the relu activation function
* Input\_dim=9 the input layer has 9 features or dimensions. This should correspond with the number of features in the vector
* Units=64 this is the number of units in the first layer. 64 was chosen by taking the average of input and output layer nodes

#Add 2nd hidden layer

model.add(Dense(32, activation=’relu’))

* 32 units in this layer, so we have fewer the deeper we go
* Activation relu using relu again

# Adding the output layer

model.add(Dense(1, activation = 'sigmoid'))

* Units = 1 because we have a binary classification problem
* Using sigmoid because we want to produce the probability of attrition

# Create optimizer with default learning rate

# Compile the model

model.compile(optimizer='SGD', loss='binary\_crossentropy', metrics=['accuracy'])

* SGD is stochastic Gradient Descent and this is the optimization algorithm
* Binary\_crossentropy is used for binary classification problems. The inputs must be 0 or 1
  + It measures the difference between the actual and predicted probabilities
* Accuracy is the evaluation metric used to assess performance. Accuracy is the number of correct predictions/total predictions

#This method stores the training process history in the variable history

history=model.fit(X\_train, y\_train,

          validation\_split=0.2,

          epochs=50,

          batch\_size=32,verbose=1)

* History object returned by fit() contains info about the training process, including the loss and performance metric at each epoch.
* Model.fit() trains the model using the training data
* Batch\_size=32 means 32 samples at a time will be processed. This is called mini batch gradient descent
* Verbose=1 is a verbose mode and controls the amount of detail that is printed. 1 is most commonly used

**Create the chart that measures loss per epoch**

# Capturing learning history per epoch

hist  = pd.DataFrame(history.history)

hist['epoch'] = history.epoch

* History.history contains all of the relevant metrics – loss, val\_loss, and accuracy
* Converts them into a columns with the df
* Also, saves the epoch number to the df which can then be used in the chart

# Plotting accuracy at different epochs

plt.plot(hist['loss'])

plt.plot(hist['val\_loss'])

plt.legend(("train" , "valid") , loc =0)

* Hist[‘loss’] Plots the loss (y-axis) against epochs (x-axis)
* Hist[‘val\_loss’] plots the validation loss at each epoch

#Printing results

results = model.evaluate(X\_test, y\_test)

* This prints two lines above the chart that provide the Test stats

**2)ADDING LAYERS TO THE BASE MODEL and OPTIMIZING WITH ADAM**

model1 = Sequential()

model1.add(Dense(256,activation='relu',kernel\_initializer='he\_uniform',input\_dim = X\_train.shape[1]))

model1.add(Dense(128,activation='relu',kernel\_initializer='he\_uniform'))

model1.add(Dense(64,activation='relu',kernel\_initializer='he\_uniform'))

model1.add(Dense(32,activation='relu',kernel\_initializer='he\_uniform'))

model1.add(Dense(1, activation = 'sigmoid'))

optimizer = tf.keras.optimizers.Adam(0.001)

model1.compile(loss='binary\_crossentropy',optimizer=optimizer,metrics=['accuracy'])

history1 = model1.fit(X\_train,y\_train,batch\_size=64,epochs=50,verbose=1,validation\_split = 0.2)

* Adds kernal\_initializer=’he\_uniform’ this is a weight initialiation method that helps the model learn faster
* Input\_dim = X\_train.shape[1] this defines the number of features in the model.
* Using the Adam optimizer with a 0.001 learning rate
* Adam combines SGD and RMSProp and is widely used because it adapts the learning rate during training

**3)ADD BATCH NORMALIZATION**

model2 = Sequential()

model2.add(Dense(128,activation='relu',input\_dim = X\_train.shape[1]))

model2.add(BatchNormalization())

model2.add(Dense(64,activation='relu',kernel\_initializer='he\_uniform'))

model2.add(BatchNormalization())

model2.add(Dense(32,activation='relu',kernel\_initializer='he\_uniform'))

model2.add(Dense(1, activation = 'sigmoid'))

optimizer = tf.keras.optimizers.Adam(0.001)

model2.compile(loss='binary\_crossentropy',optimizer=optimizer,metrics=['accuracy'])

history\_2 = model2.fit(X\_train,y\_train,batch\_size=64,epochs=50,verbose=1,validation\_split = 0.2)

**4)ADD DROPOUT**

model3 = Sequential()

model3.add(Dense(256,activation='relu',input\_dim = X\_train.shape[1]))

model3.add(Dropout(0.2))

model3.add(Dense(128,activation='relu'))

model3.add(Dropout(0.2))

model3.add(Dense(64,activation='relu'))

model3.add(Dropout(0.2))

model3.add(Dense(32,activation='relu'))

model3.add(Dense(1, activation = 'sigmoid'))

optimizer = tf.keras.optimizers.Adam(0.001)

model3.compile(loss='binary\_crossentropy',optimizer=optimizer,metrics=['accuracy'])

history\_3 = model3.fit(X\_train,y\_train,batch\_size=64,epochs=50,verbose=1,validation\_split = 0.2)

**5)KERAS OPTIMIZER**

#architecture

model.add(Dense(256, activation='relu', input\_dim=X\_train.shape[1]))

model.add(Dropout(0.3))

model.add(Dense(128, activation='relu'))

model.add(Dropout(0.3))

model.add(Dense(64, activation='relu'))

model.add(Dropout(0.2))

model.add(Dense(32, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

# Set optimizer to Adam

optimizer = tf.keras.optimizers.Adam()

# Compile the model with binary cross-entropy loss and accuracy metric

model.compile(optimizer=optimizer, loss='binary\_crossentropy', metrics=['accuracy'])

# Train the model

history = model.fit(X\_train, y\_train, epochs=50, batch\_size=64, verbose=1, validation\_split=0.2)

**Using Grid Search**

# Wrap the model into a KerasClassifier for Scikit-learn compatibility. This allows us to use Scikit-learn functions (GridSearchCV) and provides us with methods like .fit(), predict(), and score().

keras\_estimator = KerasClassifier(build\_fn=lambda: model, verbose=1)

# Define the grid search parameters

learn\_rate = [0.01, 0.1, 0.001]

batch\_size = [32, 64, 128]

param\_random = dict(optimizer\_\_learning\_rate=learn\_rate, batch\_size=batch\_size)

# Perform Randomized Search

random\_search = RandomizedSearchCV(estimator=keras\_estimator, param\_distributions=param\_random, cv=3, verbose=1, n\_jobs=-1)

# Fit the RandomizedSearchCV

random\_search.fit(X\_train, y\_train)

# Print the best result

print("Best: %f using %s" % (random\_search.best\_score\_, random\_search.best\_params\_))

model.summary()

# Evaluate the model on the test set

test\_loss, test\_acc = model.evaluate(X\_test, y\_test, verbose=1) print('Test accuracy:', test\_acc)

**Five steps for the binary classification NN code**

1)Build Architecture

* Number of nodes for input, hidden, and output layers
* Activation function for hidden
* Drop out for hidden
* Define output function – Sigmoid or Softmax

2)Compile the code

* Compile loss – cross entropy is common
* Optimizer – Adam
* Metrics - accuracy

3)Run the model

* Train
* Validation split – is not the test. There are three chunks of data for NN – train, validation, and test
* Epochs – how many iterations
* Verbose

4)Create the visualizations

* Measure loss per epoch for training and validation data
* Measure accuracy per epoch
* ROC Curve

5)Confusion Matrix

* Score the predicted probabilities as a 0 or 1

**Build Architecture**

model = Sequential()

model.add(Dense(activation = 'relu', input\_dim = 9, units=128))

model.add(Dense(64, activation=’relu’))

model.add(Dense(32, activation=’relu’))

model.add(Dense(1, activation = 'sigmoid'))

**Compile Code**

model.compile(optimizer='SGD', loss='binary\_crossentropy', metrics=['accuracy'])

**Run the Model**

history=model.fit(X\_train, y\_train, validation\_split=0.2, epochs=50,

          batch\_size=32,verbose=1)

**Visualizations – Loss/Accuracy by Epoch and ROC Curve**

**Loss/Accuracy**

hist = pd.DataFrame(history.history) #y-axis vars

hist['epoch'] = history.epoch #x-axis

plt.plot(hist['loss'])

plt.plot(hist['val\_loss'])

plt.legend(("train" , "valid") , loc =0)

hist = pd.DataFrame(history.history) #y-axis vars

hist['epoch'] = history.epoch #x-axis

plt.plot(hist['accuracy'])

plt.plot(hist['val\_accuracy'])

plt.legend(("train" , "valid") , loc =0)

**ROC Curve**

yhat1 = model1.predict(X\_test) # predict probabilities

yhat1 = yhat1[:, 0] # keep probabilities for the positive outcome only

fpr, tpr, thresholds1 = roc\_curve(y\_test, yhat1) # calculate roc curves

gmeans1 = np.sqrt(tpr \* (1-fpr)) # calculate the g-mean for each threshold

ix = np.argmax(gmeans1) # locate the index of the largest g-mean

print('Best Threshold=%f, G-Mean=%.3f' % (thresholds1[ix], gmeans1[ix]))

# plot the roc curve for the model

pyplot.plot([0,1], [0,1], linestyle='--', label='No Skill') #diagonal line

pyplot.plot(fpr, tpr, marker='.') #plots the ROC curve

pyplot.scatter(fpr[ix], tpr[ix], marker='o', color='black', label='Best')

# axis labels

pyplot.xlabel('False Positive Rate')

pyplot.ylabel('True Positive Rate')

pyplot.legend()

# show the plot

pyplot.show()

**Confusion Matrix**

y\_pred\_test = model.predict(X\_test) #loads training model and applies to test

y\_pred\_test = (y\_pred > 0.5) #automatically scores gt 0.5 to 1 and lt 0

def metrics\_score(actual, predicted):

  print(classification\_report(actual, predicted))

  cm = confusion\_matrix(actual, predicted)

  plt.figure(figsize=(8,5))

  sns.heatmap(cm, annot=True,  fmt='.2f', xticklabels=['Not Attrite', 'Attrite'], yticklabels=['Not Attrite', 'Attrite'])

  plt.ylabel('Actual')

  plt.xlabel('Predicted')

  plt.show()

metrics\_score(y\_test, y\_pred\_test)

**Recommendation Systems** [**@**](#_top)

Recommendation system is a matching engine. Recommendation system’s objective is to satisfy your search criteria with an ordered list of recommendations, be it food, movies, songs, news, clothes, books, people, etc.

**Prediction Problem**

It is our job as a data scientist to determine the likelihood of matching a user to a provider/item at a given time and a given context.

N users (rows=i) and M providers or items (columns=j). We want to find

Lij which is the likelihood of matching user to item. Think of it as a large matrix and the cells include the value for Lij. How likely will user A like item B.

**Recommendation System Methods**

Recommendation systems start with the user-item matrix. Users as rows and items as columns. Most matrices are largely empty so these empty values need to be filled in.

**Averaging** – assume all users are the same and just insert the average, e.g., if the average for movie x is 3.25, we insert 3.25 into all missing values. This is also called a simple ranked based recommendation system based on the average.

Data is the user-item interaction matrix. The match is easy because it is the item average for all users

**Content Based Filtering –** uses additional information uses pre defined demos or attributes, like gender, age, or other personal information to create a more meaningful average. It can also include information about the movie, such as genre, cast, etc. The goal is to build a model based on the available features that help to explain the observed user-item interactions. For example, younger females tend to give higher ratings, and males ratings are specific to genre, so can we use this information to improve our average rating estimates.

The table is the attributes as columns and movies as rows. The match can only be done if we know something about the user in terms of demos and attribute preferences. If we do not know this information, you have to have the user input it and then match.

**Clustering -**  start by using PCA to combine items. Then use K-means to cluster people based on the item PCA. Those within a cluster would receive similar recommendations. This is considered a weak method because it does not handle those on the cluster boundaries very well.

The table is items as the columns and people as the rows. The match happens by recommending the same things with cluster groups.

**Collaborative Filtering -**  this is known as personalized clustering and is one of the most popular approaches. It focuses not on predefined attributes but behaviors based largely on likes and dislikes.

* Filter out users and items with not enough ratings
* Similarity Calculation from the user-item matrix. This calculates

User-User and Item-Item Collaborative Filtering are the two most common methods.

* User-user defines a subset of users, and then recommends items that this subset rates highly. User-user is about the similarity of users. Item-Item is a subset of items based on ALL users and recommends items based on the similarity of items.
* User-User Collaborative Filtering is a technique used to predict the items that a user might like based on the ratings given to items by other **users who have similar tastes to the target user.**
* User-User uses ratings from similar users to predict the items a user might like. It we just focus on people like you, and you cohort ranks movie A and movie B highly, then you will likely be recommended movie A and B.
* Item-Item Collaborative Filtering is a technique used to predict the items that a user likes based on finding **similarities between items that the user had rated and the target item.**
* Item-Item uses user ratings from other products to predict the items a user may like. Thus if most people who like movie A also like movie B, and you rate movie A highly, you will be recommeded movie B. This is item-item because we are looking across ALL users and a subset of items.

**Matrix Factorization –** collaborative filtering can fall short if the data is too sparse. Matrix factorization can help. It works by representing users and items in a lower dimensional space and extract hidden features. These hidden features are also called latent features, just like Factors.

Data is items as columns and users as rows

**Singular Value Decomposition (SVD)** - is one of the common technigues

SVD decomposes the original matrix X into three smaller matrix

X = USVt

X = the original user-item interaction matrix size m x n

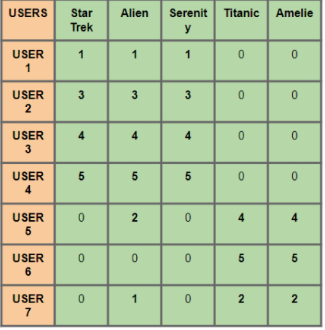
U = Matrix of Latent Features for Users with size m x r.

Vt= Matrix of Latent Features for items with size r x n.

S = It is a diagnoal matrix of singular values with size r x r. This creates the importance of the latent Factors (PCA). The row and column count will be the same as the number of PCA dimensions

The shape of U and V must be m x r and r x n because of matrix mulitplcation. The number of columns in the first matrix must equal the number of rows in the second matrix.

X is below. It is the original user-item interaction matrix 5 x 7



U = Matrix of Latent Features for Users with size 3 x 7. This matrix uses PCA to reduce the dimnsionality of the items.

A screenshot of a table

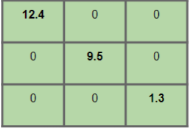
Description automatically generated

Vt= Matrix of Latent Features for items with size r x n. This matrix uses PCA or to reduce the dimensionality of users. The number of columns in U must be the same as the number of rows in V so we can multiply the two matrices. The output is S below

**A grid of numbers and symbols

Description automatically generated with medium confidence**

S = It is a diagnoal matrix of singular values with size r x r. Each value in the diagonal represents the ith singular value which indicates the strength of the ith latent feature. Multiplying X by Xt would give us a 5x5 matrix but we are only interested in the top 3 which aligns with number of PCA dimensions defined in both matrices U and Vt

****

We can rebuild matrix X using these three subsets: X = USVt

**CODE**

**1) Data prep – sparse data so generally need to remove items and people with minimal reviews**

**2) Build models**

**3) Find the books, movies, etc to recommend**

**4) Evaluate performance of models**

**TERMINOLOGY**

* Relevant item: An item rated higher than the threshold (ACTUAL)
* Recommended item: An item predicted to be higher than the threshold (PREDICTED)
* FN is the frequency of relevant that are NOT recommended to the user
* FP is the frequency of recommended items that ar NOT relevant
* Recall is the fraction of relevant items that are recommended to the user. Recall minimizes FN and if focused on actual accuracy
  + Recommended(predicted)/Relevant(actual)
  + If out of 10 relevant products, 6 are recommended, that is 60%
* Precision is the fraction of recommended items that are relevant. Precision minimizes FP and is focused on forecasting accuracy.
  + Relevant(actual)/Recommended(predicted)
  + If out of 10 recommended items, 6 are relevant, then precision is 60%

**Library’s**

# To compute the accuracy of models

from surprise import accuracy

# Class is used to parse a file containing ratings, data should be in structure - user ; item ; rating

from surprise.reader import Reader

# Class for loading datasets

from surprise.dataset import Dataset

# For tuning model hyperparameters

from surprise.model\_selection import GridSearchCV

# For splitting the rating data in train and test datasets

from surprise.model\_selection import train\_test\_split

# For implementing similarity-based recommendation system

from surprise.prediction\_algorithms.knns import KNNBasic

# For implementing matrix factorization based recommendation system

from surprise.prediction\_algorithms.matrix\_factorization import SVD

# for implementing K-Fold cross-validation

from surprise.model\_selection import KFold

# For implementing clustering-based recommendation system

from surprise import CoClustering

**1) Filter out those with less than 50 reviews**

# Step 1: Filter users with 50+ reviews

user\_counts = df.groupby('user\_id')[‘rating’].count()

* User\_id is the index and count is the column in this series

filtered\_users = df[df['user\_id'].isin(user\_counts[user\_counts >= 50].index)]

* isin(user\_counts[user\_counts >= 50].index)] is a boolean that filters out those with lt 50.
* The .index resets the index so user id is in the new df
* This is just like an inner join between userids with more than 50 reviews and the original df. We could also do a pd.merge with an inner join to achieve the same result.

# Step 2: Filter books with 10+ reviews

book\_counts = filtered\_users.groupby('book\_id')[‘rating’].count()

filtered\_books = filtered\_users[filtered\_users['book\_id'].isin(book\_counts[book\_counts >= 10].index)]

**2) Recommendation system models**

**Model 1: Simple Rank Based System based on the rating average**

# Calculating average ratings

average\_rating = df.groupby('book\_id')['rating'].mean()

# Calculating the count of ratings

count\_rating = df.groupby('book\_id')['rating'].count()

# Making a dataframe with the count and average of ratings

final\_rating = pd.DataFrame({'avg\_rating':average\_rating, 'rating\_count':count\_rating})

**Function to find the top n items for the rank based system**

def top\_n\_books(data, n, min\_interaction=100):

    # Finding books with minimum number of interactions

    recommendations = data[data['rating\_count'] > min\_interaction]

    # Sorting values w.r.t. average rating

recommendations = recommendations.sort\_values(by='avg\_rating', ascending=False)

    return recommendations.index[:n]

def top\_n\_books(data, n, min\_interaction=100):

* Min\_interactions=100 is an optional parameter. It specifies the minumum number of user reviews

recommendations = data[data['rating\_count'] > min\_interaction]

* Creates a df called recommendations
* It only returns rows where the ‘rating\_count > 100

recommendations.sort\_values(by='avg\_rating', ascending=False)

* This sorts the recommendations df by avg rating in descending order so the largest scores are at the top

return recommendations.index[:n]

* This returns the index which in this case is the book id
* [:n] slices the index to return only the first n items

**Call the function and build a list**

# list the top 5 books with 10 minimum interactions

Calling the previous function top\_n\_books within this function. The function above filters on interactions greater than the parameter, sorts the df by avg rating, and then selects the top n rows. The function is then inserted into list below.

res = list(top\_n\_books(final\_rating, 5, 10))

# Name of the books

list\_of\_books = []

for i in res:

    list\_of\_books.append(df[df['book\_id']== str(i) ]['Book-Title'].unique()[0])

list\_of\_books

**calling the function above**

res = list(top\_n\_books(final\_rating, 5, 10))

* Converts top\_n\_books to a list – top 5 books with at least 10 reviews

list\_of\_books = []

* This is an empty list that will contain the name of the top n books

for i in res:

* This loops through each element in the res list. i is the current book\_id

list\_of\_books.append(df[df['book\_id']== str(i) ]['Book-Title'].unique()[0])

* df['book\_id']== str(i) checks to see if book\_id from the df matches the current value of i
* df[df['book\_id']== str(i) ] this filters the df to only include rows where the book id matches i
* Each book is added to the list by the append command

**Model 2: Collaborative Filtering User-User**

**1)Data Prep**

#Label encoding converts user\_id and book\_id into a numerical value, eg, userid is changed from 4665555K to 1.

from sklearn.preprocessing import LabelEncoder

data=df[['user\_id','book\_id']].apply(LabelEncoder().fit\_transform)

data['rating']=df['rating']

data.head()

**Create Table with prod\_id, avg\_rating, and rating count**

#Calculating average ratings

average\_rating = data.groupby('book\_id')['rating'].mean()

#Calculating the count of ratings

count\_rating = data.groupby('book\_id')['rating'].count()

#Updating the final\_rating dataframe with the new encoded book\_id count and average of ratings based on the new dataframe

final\_rating = pd.DataFrame({'avg\_rating':average\_rating, 'rating\_count':count\_rating})

final\_rating.head()

**2)Convert dataset into a surprise dataset and split into train and test**

# Instantiating Reader scale with expected rating scale. This helps helps with interpreting the results

reader = Reader(rating\_scale=(1, 10))

# Loading the rating dataset into a format the used by surprise

data = Dataset.load\_from\_df(data[['user\_id', 'book\_id', 'rating']], reader)

# Splitting the data into train and test dataset

trainset, testset = train\_test\_split(data, test\_size=0.3, random\_state=42)

**3)Run the model**

sim\_options = {'name': 'cosine',

               'user\_based': True}

algo\_knn\_user = KNNBasic(sim\_options=sim\_options,verbose=False)

# Train the algorithm on the train set, and predict ratings for the test set

algo\_knn\_user.fit(trainset)

**4)Run the evaluation function**

See 7) function. The code calls the function - knn\_user is the model

precision\_recall\_at\_k(knn\_user)

**5)Find predicted rating for a specific user**

algo\_knn\_user.predict(1326, 12126, r\_ui=8, verbose=True)

**6)Using the same user, predict the rating for a book they have not rated**

algo\_knn\_user.predict(1326, 2150, verbose=True)

**7)Evaluate the performance of a recommendation system**

def precision\_recall\_at\_k(model, k = 10, threshold = 3.5):

    """Return precision and recall at k metrics for each user"""

    # First map the predictions to each user

    user\_est\_true = defaultdict(list)

    # Making predictions on the test data

    predictions = model.test(testset)

    for uid, \_, true\_r, est, \_ in predictions:

        user\_est\_true[uid].append((est, true\_r))

    precisions = dict()

    recalls = dict()

    for uid, user\_ratings in user\_est\_true.items():

        # Sort user ratings by estimated value

        user\_ratings.sort(key = lambda x: x[0], reverse = True)

        # Number of relevant items

        n\_rel = sum((true\_r >= threshold) for (\_, true\_r) in user\_ratings)

        # Number of recommended items in top k

        n\_rec\_k = sum((est >= threshold) for (est, \_) in user\_ratings[:k])

        # Number of relevant and recommended items in top k

        n\_rel\_and\_rec\_k = sum(((true\_r >= threshold) and (est >= threshold))

                              for (est, true\_r) in user\_ratings[:k])

        # Precision@K: Proportion of recommended items that are relevant

        # When n\_rec\_k is 0, Precision is undefined. Therefore, we are setting Precision to 0 when n\_rec\_k is 0

        precisions[uid] = n\_rel\_and\_rec\_k / n\_rec\_k if n\_rec\_k != 0 else 0

        # Recall@K: Proportion of relevant items that are recommended

        # When n\_rel is 0, Recall is undefined. Therefore, we are setting Recall to 0 when n\_rel is 0

        recalls[uid] = n\_rel\_and\_rec\_k / n\_rel if n\_rel != 0 else 0

    # Mean of all the predicted precisions are calculated.

    precision = round((sum(prec for prec in precisions.values()) / len(precisions)), 3)

    # Mean of all the predicted recalls are calculated.

    recall = round((sum(rec for rec in recalls.values()) / len(recalls)), 3)

    accuracy.rmse(predictions)

    print('Precision: ', precision) # Command to print the overall precision

    print('Recall: ', recall) # Command to print the overall recall

    print('F\_1 score: ', round((2\*precision\*recall)/(precision+recall), 3)) # Formula to compute the F-1 score

**Model 3: Collaborative Filtering User-User using Grid Search**

# Setting up parameter grid to tune the hyperparameters

param\_grid = {'k': [20, 40], 'min\_k': [3, 6, 9],

              'sim\_options': {'name': ['msd', 'cosine','pearson'],

              'user\_based': [True]}}

# Performing 3-fold cross-validation to tune the hyperparameters

gs = GridSearchCV(KNNBasic, param\_grid, measures=['rmse', 'mae'], cv=3, n\_jobs=-1)

# Fitting the data

gs.fit(data)

# Best RMSE score

print(gs.best\_score['rmse'])

# Combination of parameters that gave the best RMSE score

print(gs.best\_params['rmse'])

output: {'k': 40, 'min\_k': 6, 'sim\_options': {'name': 'cosine', 'user\_based': True}}

**Run model with optimal parameters**

# Using the optimal similarity measure for user-user based collaborative filtering

sim\_options = {'name': 'cosine',

               'user\_based': True}

# Creating an instance of KNNBasic with optimal hyperparameter values

knn\_user\_opt = KNNBasic(sim\_options=sim\_options, k=40, min\_k=6, verbose=False)

# Training the algorithm on the trainset

knn\_user\_opt.fit(trainset)

# Let us compute precision@k and recall@k also with k =10

precision\_recall\_at\_k(knn\_user\_opt)

**Get recommendations function**

def get\_recommendations(data, user\_id, top\_n, algo):

    # Creating an empty list to store the recommended product ids

    recommendations = []

    # Creating an user item interactions matrix

    user\_item\_interactions\_matrix = data.pivot(index = 'user\_id', columns = 'prod\_id', values = 'rating')

    # Extracting those product ids which the user\_id has not interacted yet

    non\_interacted\_products = user\_item\_interactions\_matrix.loc[user\_id][user\_item\_interactions\_matrix.loc[user\_id].isnull()].index.tolist()

    # Looping through each of the product ids which user\_id has not interacted yet

    for item\_id in non\_interacted\_products:

        # Predicting the ratings for those non interacted product ids by this user

        est = algo.predict(user\_id, item\_id).est

        # Appending the predicted ratings

        recommendations.append((item\_id, est))

    # Sorting the predicted ratings in descending order

    recommendations.sort(key = lambda x: x[1], reverse = True)

    return recommendations[:top\_n] # Returing top n highest predicted rating products for this user

**Call function above and list out the recommended products**

# Making top 5 recommendations for user\_id "A3LDPF5FMB782Z" with a similarity-based recommendation engine

recommendations = get\_recommendations(df, "A3LDPF5FMB782Z", 5, knn\_user\_opt)

# Building the dataframe for above recommendations with columns "prod\_id" and "predicted\_ratings"

pd.DataFrame(recommendations, columns=['prod\_Id', 'predicted\_ratings'])

**Model 3 Collaborative Filtering Item-Item**

# Declaring the similarity options

sim\_options = {'name': 'cosine',

               'user\_based': False}

# KNN algorithm is used to find desired similar items. Use random\_state=1

knn\_item = KNNBasic(sim\_options=sim\_options,verbose=False)

# Train the algorithm on the trainset, and predict ratings for the test set

knn\_item.fit(trainset)

# Let us compute precision@k, recall@k, and f\_1 score with k = 10

precision\_recall\_at\_k(knn\_item)

**Model 4 Matrix-Factorization SVD**

# Using SVD matrix factorization. Use random\_state = 1

svd = SVD(random\_state=1)

# Training the algorithm on the trainset

svd.fit(trainset)

# Use the function precision\_recall\_at\_k to compute precision@k, recall@k, F1-Score, and RMSE

precision\_recall\_at\_k(svd)

**Model 5 Matrix-Factorization SVD with Grid Search**

# Set the parameter space to tune

param\_grid = {'n\_epochs': [10, 20, 30], 'lr\_all': [0.001, 0.005, 0.01],

              'reg\_all': [0.2, 0.4, 0.6]}

# Performing 3-fold gridsearch cross-validation

gs\_vsd = GridSearchCV(SVD, param\_grid, measures=['rmse'], cv=3, n\_jobs=-1)

# Fitting data

gs\_vsd.fit(data)

# Best RMSE score

print(gs\_vsd.best\_score['rmse'])

# Combination of parameters that gave the best RMSE score

print(gs\_vsd.best\_params['rmse'])

**Build the optimized SVD model using optimal hyperparameter search**

Use random\_state=1

svd\_opt = SVD(n\_epochs=20, lr\_all=0.01, reg\_all=0.2, random\_state=1)

# Train the algorithm on the trainset

svd\_opt=svd\_opt.fit(trainset)

# Use the function precision\_recall\_at\_k to compute precision@k, recall@k, F1-Score, and RMSE

precision\_recall\_at\_k(svd\_opt)

**Data Structures** [**@**](#_top)

**Array**

A variable that can hold more than one variable at a time. A list is an array in Python

Instead of creating 10 variables to hold the ages of 10 students, we create a list and store all 10 in one list, aka one variable. We access the list by indexing.

Arrays in python are efficient when fast lookups are required but the list does not change much. They are inefficient when it comes to adding or removing elements. If the list will change frequently, then a linked list is more efficient

**Linked List**

Is a linear data structure with two parts – data and reference. Data holds data and reference holds node information.

Linked list are efficient when a list needs to be altered frequently. The nodes are linked so when inserting or deleting an element we don’t need to shift all of the elements like an array, it just adjust the reference or node information.

A linked list is like a dictionary with a key and a value. They key is the data and the value is the reference to the next node. This structure relates one node to the next linearly