**Dealing with missing values**

Three main strategies:

* Drop columns/rows with missing values
* Imputation: filling the missing values with some number
* Impute missing values and add an additional column where the entries show which values have been imputed

Removing the columns with null values

**df[‘column’].dropna(axis==1)**

columns\_missing = [col for col in X\_train.columns if X\_train[col].isnull().any()]

reduced\_X\_train = X\_train.drop(columns\_missing,axis=1)

reduced\_X\_test = X\_test.drop(columns\_missing,axis=1)

**Imputation**

from sklearn.impute import SimpleImputer

imputer = SimpleImputer()

imputed\_X\_train = pd.DataFrame(imputer.fit\_transform(X\_train),columns=X\_train.columns)

imputed\_X\_test = pd.DataFrame(imputer.transform(X\_test),columns=X\_test.columns)

**Imputation as well as marking values that have been imputed**

X\_train\_plus = X\_train.copy()

X\_test\_plus = X\_test.copy()

for col in columns\_missing:

    X\_train\_plus[col+'\_is\_missing'] = X\_train\_plus[col].isnull().astype(int)

    X\_test\_plus[col+'\_is\_missing'] = X\_test\_plus[col].isnull().astype(int)

X\_train\_plus = pd.DataFrame(imputer.fit\_transform(X\_train\_plus),columns=X\_train\_plus.columns)

X\_test\_plus = pd.DataFrame(imputer.transform(X\_test\_plus),columns=X\_test\_plus.columns)

Upon running the model predictions for these approaches, the **imputation** method is found to result in the best performance.

**Categorical Variables**

Approaches to dealing with categorical labels:

* Drop columns with categorical labels: Only to be done when they don’t carry any meaningful information
* **Ordinal encoding :** Assign unique integer to each class label. To be used for ordinal variables. (**using ordinal encoder)**
* **One hot encoding :** For nominal variables, does not assume order of categories. Doesn’t perform well when there are large number of variables

**Cardinality**

The number of unique entries of a categorical variable is called the cardinality.

cardinal\_cols = [col for col in X\_train.columns if X\_train[col].nunique() < 10 and X\_train[col].dtype == "object"]

**Checking for categorical columns**

categorical\_cols = [col for col in X\_train\_f.columns if X\_train\_f[col].dtype == 'object']

categorical\_cols

**Dropping columns with categorical values**

drop\_X\_train = X\_train\_f.select\_dtypes(exclude='object')

drop\_X\_test = X\_test\_f.select\_dtypes(exclude='object')

print('The score when the categorical variables are dropped: ',score\_model(drop\_X\_train,drop\_X\_test,y\_train,y\_test))

**Using nominal encoder**

from sklearn.preprocessing import OrdinalEncoder

ordinal\_encoder = OrdinalEncoder()

label\_X\_train = X\_train\_f.copy()

label\_X\_test = X\_test\_f.copy()

label\_X\_train[categorical\_cols] = ordinal\_encoder.fit\_transform(X\_train\_f[categorical\_cols])

label\_X\_test[categorical\_cols] = ordinal\_encoder.transform(X\_test\_f[categorical\_cols])

print('The score after label encoding is: ', score\_model(label\_X\_train,label\_X\_test,y\_train,y\_test))

It is convenient if we have the categorical columns extracted beforehand so that we can transform only those columns

**The issue with new values appearing in test sets:**

Fitting an ordinal encoder to a column in the training data creates a corresponding integer-valued label for each unique value that appears in the training data. In the case that the validation data contains values that don't also appear in the training data, the encoder will throw an error, because these values won't have an integer assigned to them.

#Column that can be safely encoded

good\_cols = [col for col in categorical\_cols if set(X\_test\_f[col].unique()).issubset(X\_train\_f[col].unique())]

#Columns that either need to be dropped or preprocessed

bad\_cols = [col for col in categorical\_cols if col not in good\_cols]

**One hot encoding**

Usually yields the best results amongst the three approaches. The scikit learn One hot encoder class is used. For large datasets with many rows, one-hot encoding can greatly expand the size of the dataset. For this reason, we typically will only one-hot encode columns with relatively low cardinality. Then, high cardinality columns can either be dropped from the dataset, or we can use ordinal encoding.

from sklearn.preprocessing import OneHotEncoder

OH\_encoder = OneHotEncoder(handle\_unknown='ignore',sparse=False)

OH\_columns\_train = pd.DataFrame(OH\_encoder.fit\_transform(X\_train\_f[categorical\_cols]))

OH\_columns\_test = pd.DataFrame(OH\_encoder.transform(X\_test\_f[categorical\_cols]))

#One hot encoding removes the index. We will add it back

OH\_columns\_train.index = X\_train\_f.index

OH\_columns\_test.index = X\_test\_f.index

#Removing the original categorical columns so that we can replace them with the OH encoded ones

numerical\_col\_train = X\_train\_f.drop(categorical\_cols,axis=1)

numerical\_col\_test = X\_test\_f.drop(categorical\_cols,axis=1)

final\_X\_train = pd.concat([numerical\_col\_train,OH\_columns\_train],axis=1)

final\_X\_test = pd.concat([numerical\_col\_test,OH\_columns\_test],axis=1)

print('Score after one-hot encoding is : ',score\_model(final\_X\_train,final\_X\_test,y\_train,y\_test))

* Unlike nominal encoder, the One hot encoder doesn’t make changes in place and hence we will need to do the necessary modifications manually.
* The indexes are removed which need to be added back.
* The original categorical columns in the dataframe must be removed so that they can be replaced by the one hot encoded versions.

**Key parameters:**

* **handle\_unknown = ‘ignore’ :**This is to take care of classes which might appear in the test set but not in the training set
* **sparse = False:** Saves a lot of memory in cases where the matrix elements are mostly zeroes.

**Scaling and Normalization**

* **Scaling :** Change range of data

**Normalization:** Change distribution of data

Difference in scales across the input variables may lead to difiiculty in labelling the target label.

**Pipelines**

This is a way of making the code cleaner and debugging easier. Specifically, a pipeline bundles preprocessing and modeling steps so you can use the whole bundle as if it were a single step.

**Step 1: Defining the preprocessing steps**

* The numerical and categorical columns have been processed separately. Alternatively, we can separately specify how each column must be processed.
* We use column transformer to bundle together different preprocessing steps

**Necessary imports**

from sklearn.compose import ColumnTransformer

from sklearn.preprocessing import OneHotEncoder

from sklearn.pipeline import Pipeline

from sklearn.impute import SimpleImputer

**Preprocessing the numerical data**

numerical\_transformer = SimpleImputer(strategy='constant')

**Preprocessing the categorical data**

categorical\_transformer  = Pipeline(steps = [('imputer',SimpleImputer(strategy='most\_frequent')),

                                     ('one\_hot',OneHotEncoder(handle\_unknown='ignore'))

                                     ])

**Bundling together the preprocessing for numerical and categorical data**

preprocessor = ColumnTransformer(transformers=[('num',numerical\_transformer,numerical\_cols),

                                               ('cat',categorical\_transformer,categorical\_cols)

                                               ])

**Step 2: Define the model**

from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n\_estimators=100,random\_state=0)

**Step 3: Creating and evaluating the model**

from sklearn.metrics import mean\_absolute\_error

my\_pipeline = Pipeline(steps = [('preprocessor',preprocessor),

                        ('model',model)

                        ])

**Training the model and obtaining predictions on the valid set**

#Preprocessing the training data and training the model

my\_pipeline.fit(X\_train,y\_train)

#Obtain predictions on the test data

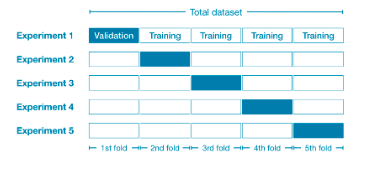
preds = my\_pipeline.predict(X\_valid)

score = mean\_absolute\_error(y\_valid,preds)

print("The model accuracy score is: ",score)

**Cross Validation**

* In general, the larger the validation set, the less randomness (aka "noise") there is in our measure of model quality, and the more reliable it will be. Unfortunately, we can only get a large validation set by removing rows from our training data, and smaller training datasets mean worse models!
* In cross-validation, we run our modeling process on different subsets of the data to get multiple measures of model quality.
* For example, we could begin by dividing the data into 5 pieces, each 20% of the full dataset. In this case, we say that we have broken the data into 5 "folds".



* During the first iteration, we hold the 1st fold as holdout while the others form the training set . As this process repeats every part of the dataset will have been the holdout at some point and we end up with a measure for model quality which takes into account all the rows in the dataset.
* Used when the dataset isn’t too large, ie training doesn’t take too long.

**Implementation**

**Defining the pipeline with preprocessing steps and model**

from sklearn.impute import SimpleImputer

from sklearn.ensemble import RandomForestRegressor

from sklearn.pipeline import Pipeline

my\_pipeline = Pipeline(steps=[

                        ('preprocessor',SimpleImputer(strategy='constant')),

                        ('model',RandomForestRegressor(n\_estimators=50,random\_state =100))])

**Instantiating the cross validation module**

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

score = -1 \* cross\_val\_score(my\_pipeline,X,y,cv=5,scoring='neg\_mean\_absolute\_error')

print('MAE scores are: ',score)

* Multiplied by -1 because sk-learn uses a **negative error term for scoring** in this case.
* **cv** parameter specifies how many fold validation we are willing to go for. 5 in this case

**Feature Engineering**

**Pandas**

**Pandas Series**

* Allows us to add on a labelled index as compared to numeric index in numpy arrays.

**Creating a series**

myser = pd.Series(data=myData,index=myIndex)

* The elements within a pandas series can be either be accessed through their numerical or their labelled index.
* Dictionaries can directly be converted to pandas series.

**Obtaining keys available in a series**

my\_series.keys()

**Operations between series**

* The operations are broadcasted. For example series\*2 will multiply every element of the series by 2
* When the addition operator is used like in the above case, the indices that aren't common between the two series will hold the value NaN.
* Alternative approach - using the add function which comes with the **fill\_value** argument which is used to specify what values to use when indices unique to a particular series are encountered.

**Pandas Dataframes**

**Creating a dataframe**

df = pd.DataFrame(data= mydata,index=myindex,columns=mycolumns)

**Getting the columns and index**

df.columns

df.index

**Working with columns**

* Accessing multiple columns:

**df[[col1,col2,.....]]**

* New columns can be created by operating upon existing columns
* Removing columns:

df.drop('tip\_percentage',axis=1)

axis - 0 : columns

1: rows

These changes aren’t permanent. To make them permanent , set **‘inplace’** to True.

**Working with rows**

* Setting one of the columns as index:

df.set\_index(‘Column\_name’)

Again, this isn’t a permanent change. Undoing this:

df.reset\_index;

The rows can be accessed in two ways: Via numeric index:

df.iloc[start\_index:end\_index]

Via labelled index:

df.loc[['row\_label1','row\_label2',....]]

**Conditional Filtering**

*df[df[‘column\_name’] >/</=/!= some\_value]]*

*df[logical condition on df[‘column]]*

* Multiple logical conditions can be combined using the AND(&) and OR(|) operators.

df[(df['tip'] > 2) & (df['smoker']== 'Yes')]

Using the i**sin** operator:

To check if the column values are in a list of available options:

df[df['day'].isin(options)]

where, **options** is a list of possible/prospective column values.

**Apply methods**

* This allows custom user functions to be applied on every value of a column

**On a single column:**

df[‘Column\_name’].apply(user\_function)

**On Multiple columns:**

1. Using lambda functions:

df[[‘col1’,’col2’]].apply(lambda df:user\_function(df[‘col1’],df[‘col2’],axis=1)

df[['total\_bill','tip']].apply(lambda df:quality(df['total\_bill'],df['tip']),axis=1)

1. Using np.vectorize

np.vectorize(user\_function)(df[‘col1’],df[‘col2’])

np.vectorize(quality)(df['total\_bill'],df['tip'])

**np.vectorize** is used to transform standard python functions into numpy aware functions. Thus these functions become computationally more efficient.

**Other useful functions**

* **Sorting a column values:**

df.sort\_values('column\_name',ascending=True)

We specify the column name that we need to sort. **ascending=False** for descending order

* **Sorting multiple column values:**

**df.sort\_values([‘col1’,’col2’])**

The values are first sorted based on col1 values and then col2.

df.sort\_values(['tip','size'])

* **Max,min values and their index:**

df[‘column’].max

df[‘column’].min

df[‘column’].idxmax

df[‘column’].idxmin

* **Counting occurrence of a column value:**

**df[‘column’].value\_counts()**

* **Getting unique values and their total count:**

**df[‘column’].unique()**

**df[‘column’].nunique()**

* **Replacing column values:**

This can be done via two approaches:

1. **Replace()**

**df[‘column].replace([list of existing values],[list of new values])**

df['sex'].replace(['Female','Male'],['F','M'])

This is easier to do when there are fewer things to replace.

1. **map()**

**df[‘column’].map(dictionary with existing value as key and new value as the corresponding value)**

mymap = {'Female':'F','Male':'M'}

df['sex'].map(mymap)

* **Duplicates:**
* **df.duplicated():** Returns True for very first instance of duplicated rows
* **df.drop\_duplicates()** - drop duplicate rows
* **Between() function**:

**df[‘column].between(start,end,inclusive=True)**

df[df['total\_bill'].between(10,20,inclusive=True)]

This can be used during conditional filtering too.

* Additional functions:

df.nsmallest(10,'tip') #obtain 10 smallest elements from tips column

df.nlargest(10,'tip') #obtain 10 largest elements from tips column

**Missing Data**

* **x is np.nan -**  Checking if a value is missing
* **Checking for null values:**

**df[‘column’].isnull()**

**df[‘column’].notnull()**

Filtering columns with no null values:

**df[df[‘column’].notnull()]**

* **Dropping null values**

**df.dropna()**

This will drop rows with any missing values. This can however be further modified through parameters.

df = df.dropna(thresh=1)

**‘thresh**’ parameter specifies the number of non-NA parameters that must exist for the row not to be dropped

df.dropna(axis=1)

To switch between dropping rows and columns

df.dropna(subset=['last\_name'])

When the **subset** parameter is used, only the columns specified in that list will be considered. In this case only rows that contain NAN values under the ‘last\_name’ column will be dropped.

* **Filling null values**

df['pre\_movie\_score'].fillna(0)

Either fill the null values with a predefined constant or use mean imputation.

df['post\_movie\_score'].fillna(np.mean(df['post\_movie\_score']))

* **Interpolation in pandas series**

airline\_tix = {'first':100,'business':np.nan,'economy-plus':50,'economy':30}

series = pd.Series(airline\_tix)

series.interpolate()

**GroupBy operations**

* Used to aggregate continuous values per category.

**Grouping by a single categorical column**

df.groupby('model\_year').mean()

Through this we obtain the mean for all the other columns in the dataframe. We can however mention the columns we are interested in:

df.groupby('model\_year').mean()['mpg']

* The categorical column values become the index and the aggregation is performed on the numeric columns

**Grouping multiple columns at a time**

df.groupby(['model\_year','cylinders']).mean()

* This gives rise to something known as multi-index, where the outer level will be the first column value whereas the inner level will be the second column value and so on.

The indices created can be accessed/viewed via the following command:

df.groupby(['model\_year','cylinders']).mean().index

year\_cyl.index.levels

Gives the index for the outer level(year) and the inner one(number of cylinders)

**Accessing the multi-indexed elements**

1. **Via df.loc**

**df.loc[[val1,val2]] :** This can be used to access the outer level index(which in the above case would be model\_year)

year\_cyl.loc[[70,82]]

To access a single row(that is for a particular outer level value and a fixed inner level value) we need to pass the tuple which can be obtained via the indexing command mentioned above.

year\_cyl.loc[(70,8)]

Here 70 is the outer level value and 8 is the inner level value.

1. **Using cross section(.xs)**

year\_cyl.xs(key=70,level='model\_year')

**level**  parameter is used to specify which index(outer or inner) we are referencing and the **key** parameter holds the value to be indexed.

year\_cyl.xs(key=6,level='cylinders')

Particularly useful for inner level index which cannot be accessed easily via the **loc** function.

Note : unlike in **loc** function ,we can’t pass in multiple index values at once. hence, extra care must be taken while grouping the original dataframe.

* Swapping the outer and inner index: **- df.swaplevel()**
* Sorting levels: **df.sort\_index(level=’column\_name’,ascending=False)**

**Specifying a group of aggregate functions**

1. On the entire dataframe:

**df.agg([‘oper1’,’oper2’,…..])**

df.agg(['std','mean'])

1. Specifying separate operations for separate columns:

df.agg(‘col1’:[‘oper1’,’oper2’],’col2’:[‘oper1’,’oper2’]})

df.agg({'mpg':['max','mean'],'weight':['mean','std']})

**Common aggregate methods**

* mean(): Compute mean of groups
* sum(): Compute sum of group values
* size(): Compute group sizes
* count(): Compute count of group
* std(): Standard deviation of groups
* var(): Compute variance of groups
* sem(): Standard error of the mean of groups
* describe(): Generates descriptive statistics
* first(): Compute first of group values
* last(): Compute last of group values
* nth() : Take nth value, or a subset if n is a list
* min(): Compute min of group values
* max(): Compute max of group values

**Combining Dataframes**

**Concatenation**

* This involves gluing together two data-frames. The dataframes to be concatenated must be in the same format (same indices or same columns)

one = pd.DataFrame(data\_one)

two = pd.DataFrame(data\_two)

pd.concat([one,two],axis=1)

**axis =0** to concatenate along the columns .

**Inner Merge (A intersection B)**

* We usually merge two or more dataframes using an ‘on’ column. This is usually the column / feature that is common between these dataframes.
* Using inner merge, we obtain the column values that are common across the dataframes. Example is df1[‘name’] = [A,B,C,D,E] and df2[‘name’] = [A,H,C,X,Y] the inner merge will yield [A,C]

pd.merge(registrations,logins,how='inner',on='name')

how = what type of merge we want to use

on = the column on which the merging must be carried out

**Left Merge**

* All the elements present in the left table will be present in the final output, irrespective of whether they are present in the right table or not.

pd.merge(left=registrations,right=logins,how='left',on='name')

**Right Merge**

* All the elements present in the right table will be present in the final output, irrespective of whether they are present in the left table or not. The ones absent in the left column will have NaN values.

pd.merge(left=registrations,right=logins,how='right',on='name')

**Outer Merge(A union B)**

pd.merge(registrations,logins,how='outer',on='name')

* This will grab all the column values from the two data frames.

**Exploring the parameters of merge function**

* While we have used the **on**  parameter to specify the column based on which the dataframes must be merged , there are other ways too:
* **left\_on** and **right\_on**  to specify the columns in the left and right dataframe based on which the merging must be carried out

results = pd.merge(registrations,logins,how='inner',left\_on='reg\_name',right\_on='name')

* **left\_index=True**  or **right\_index=True**  in case we need to merge based on the index values of the dataframes (usually employed when **set\_index(‘column’)**  has been used)

pd.merge(left=registrations,right=logins,left\_index=True,right\_on='name',how='inner')

* In case the dataframes being merged have a column with the same name, these are automatically labelled in the resulting data frame as **column\_x and column\_y.**
* We can however specify the suffixes to be used for these columns:

pd.merge(registrations,logins,on='name',how='inner')

**Text Methods for String Data**

* The string methods that can be applied on a normal python string can be applied on python strings using **pandas\_object.str.method**

Example: Splitting the strung and expanding it into a dataframe

tech\_finance = ['GOOG,APPL,AMZN','JPM,BZC,GS']

tickers = pd.Series(tech\_finance)

tickers.str.split(',',expand=True)

Cleaning feature values(string type)

messy\_names = pd.Series(['andrew  ','bo;bo','   claire   '])

As we can notice, the names are messy in the sense they have spaces or special characters which make it hard to process them. Based on the requirement, we can use a combination of string functions to clean these.

messy\_names = messy\_names.str.replace(';','').str.strip().str.capitalize()

Alternative would be use apply calls with our customized functions:

def cleanup(name):

    name= name.replace(';','')

    name = name.strip()

    name = name.capitalize()

    return name

messy\_names.apply(cleanup)

* It is observed that using apply calls with customized functions(preferably customized) are faster than the pandas in built string calls.

**CSV Files**

Reading from csv files:

df = pd.read\_csv('example.csv')

**header=None** if you don't want the first row to be set as the header.

Writing to csv files:

df.to\_csv('new\_file.csv',index=False)

**index=False** if you don’t want the index to be saved.

**HTML tables**

df = pd.read\_csv(url)

The **‘url’** variable must contain the url of the page from where we seek to obtain our data

Writing to csv files:

df.to\_html('new\_file.html',index=False)

**Excel Files**

* Pandas treats excel files as dictionaries with the key being sheet names and the values being the dataframe representing the sheet itself.
* Ideally, when dealing with excel files it is recommended to go with the libraries (openpyxl and xlrd) instead of Pandas

**Reading Excel Files**

df = pd.read\_excel('my\_excel\_file.xlsx',sheet\_name='First\_Sheet')

In case we don't pass the sheet\_name parameter in the excel command, we will get a dictionary instead of a dataframe.

excel\_sheet\_dict = pd.read\_excel('my\_excel\_file.xlsx',sheet\_name=None)

In this case **excel\_sheet\_dict** will be a dictionary and the sheet can be accessed by indexing via the sheet name.

sample\_df = excel\_sheet\_dict['First\_Sheet']

**Obtaining the sheetnames**

wb = pd.ExcelFile('my\_excel\_file.xlsx')

wb.sheet\_names

**Saving to excel file:**

sample\_df.to\_excel('example.xlsx',sheet\_name='First\_Sheet',index=False)

**Pivot Tables**

**Matplotlib**

**Basic Commands**

* Pretty much self explanatory

plt.title('String Title')

plt.xlabel('x-axis')

plt.ylabel('y-axis')

plt.xlim(0,6)

plt.ylim(0,16)

**Figure and axes**

* This is an object-oriented approach to plotting. We create a ‘figure’ object which is basically like a blank canvas upon which we proceed with our plotting.

fig = plt.figure(figsize=(2,2),dpi=200)

* Once we have this figure, we can have axes on it. Axes are the plots -> Figure is the canvas.
* figsize = (width,height) dpi – clarity of the plot produced

axes1 = fig.add\_axes([0,0,1,1])

axes1.plot(a,b)

* [0,0,1,1] – Here the first two co-ordinates specify where the left corner should be. (0,0) means the axes will start at the left corner . If it was instead (0.5,0.5) the axes would have started midway through the x and y -axis
* 1,1 – This stands for the width and height. These quantities are relative to the canvas size. (1,1) means width and height are same as the figure. (0.5,0.5) would result in an axes with height and width half the size of the figure.

axes1.set\_xlabel('A')

axes1.set\_ylabel('B')

axes1.set\_xlim(0,12)

axes1.set\_ylim(0,10000)

axes1.set\_title('axis1')

**axes.set\_property() :** property – xlabel,ylabel,title,xlim,ylim

fig.savefig('new\_fig.png',bbox\_inches = 'tight' )

* Bbox\_inches parameter is used so that the x and y-axis are saved too

**Subplots using axes and figures**

fig = plt.figure()

plot1 = fig.add\_axes([0,0,0.5,1]) #We are ensuring that the first plot takes exactly half the area

plot1.set\_title('plot1')

plot2 = fig.add\_axes([0.5,0,0.5,1])

plot2.set\_title('plot2') #We are ensuring that the second plot takes up the remaining area

**Subplots**

* Using figures and axes to get subplots can get tedious and hence we go for the subplots function

fig,axes = plt.subplots(nrows =3,ncols=1)

* Here the key thing is the method how we access the subplots. Similar to how we access an element in an n-dimensional array.

#Accessing the plots (similar to accessing the element in an n-dimensional array)

axes[0].plot(x,y)

axes[1].plot(a,b)

axes[2].plot(b,y)

plt.tight\_layout() #Makes sure no axes overlap automatically

axes[0].set\_title('X vs Y')

axes[1].set\_title('A vs B')

axes[2].set\_title('B vs Y')

* **plt.tight\_layout()** makes sure that there is no overlap between the subplots
* In case we had a 2X2 subplot . In that case accessing the elements would be in the following way:

fig,axes = plt.subplots(nrows=2,ncols=2)

axes[0][0].plot(x,y)

axes[0][1].plot(a,b)

axes[1][0].plot(x,y)

axes[1][1].plot(a,b)

**Legends**

fig = plt.figure()

ax = fig.add\_axes([0,0,1,1])

#Adding labels in every plot call and using ax.legend in the end

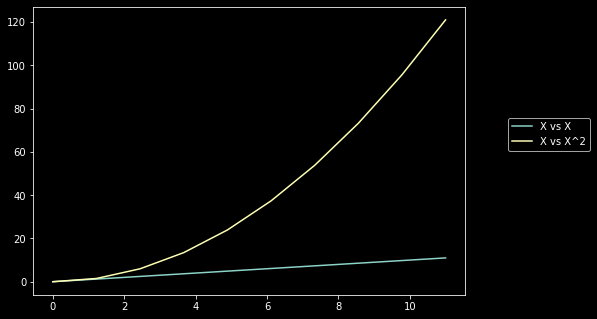
ax.plot(x,x,label= 'X vs X')

ax.plot(x,x\*\*2,label = 'X vs X^2')

#ax.legend(loc='upper left') # We can specify different locations

ax.legend(loc=(1.1,0.5)) #Again the tuple values are relative to the axes dimensions

* We add labels to every plot call and ultimately use the **ax.legend()** function. The location parameter is again relative to the underlying canvas/figure.



**Visual Styling**

* **Colour and Linewidth**

fig = plt.figure()

ax = fig.add\_axes([0,0,1,1])

ax.plot(x,x,color='#65c79b', lw=10,label='X vs X') #RGB HEX code

ax.plot(x,x+1,color='#81bccc',label= 'X vs X+1')

ax.legend()

As seen here, we can provide hex color codes . **lw** is the parameter for linewidth.

* **Linestyles**

fig = plt.figure()

ax = fig.add\_axes([0,0,1,1])

ax.plot(x,x,color='#65c79b',linewidth=4,linestyle= '-.')

We can either use the predefined linestyles or feed our customized styles.

lines = ax.plot(x,x,color='#65c79b',linewidth=4,linestyle= '-.')

lines[0].set\_dashes([5,2,5,2])

* **Markers**

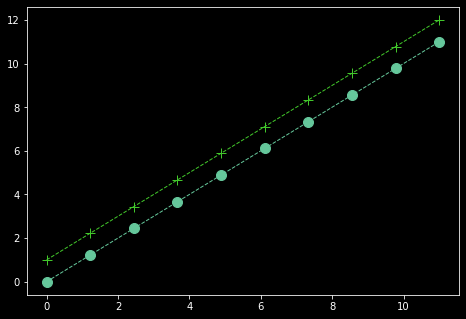
These are used to highlight the individual data points which might otherwise be masked in the graph. We can choose different markers by feeding in different string characters. Marker width is controlled by the parameter **ms.**

fig = plt.figure()

ax = fig.add\_axes([0,0,1,1])

ax.plot(x,x,color='#65c79b',linewidth=1,marker='o',ls='--',ms=10)

ax.plot(x,x+1,color='#42c72b',linewidth=1,marker='+',ls='--',ms=10)



**Seaborn**

**Scatterplots**

* Used for continuous data.

plt.figure(figsize=(5,5),dpi=200)

sns.scatterplot(x='salary',y='sales',data=df,hue='level of education',palette='Dark2')

* **‘hue’** parameter is used to colour the datapoints based on the value of the column that is fed to this parameter. If the column has a continuous feature then the resulting colouring will be a gradient and if it is categorical we will have a legend with a distinct colour for each feature
* **Palette** parameter is used to set colormaps. This is basically the colour schemes that will be adapted when the hue parameter is used.
* Similar to hue, we can vary the size of datapoints based on certain column values.

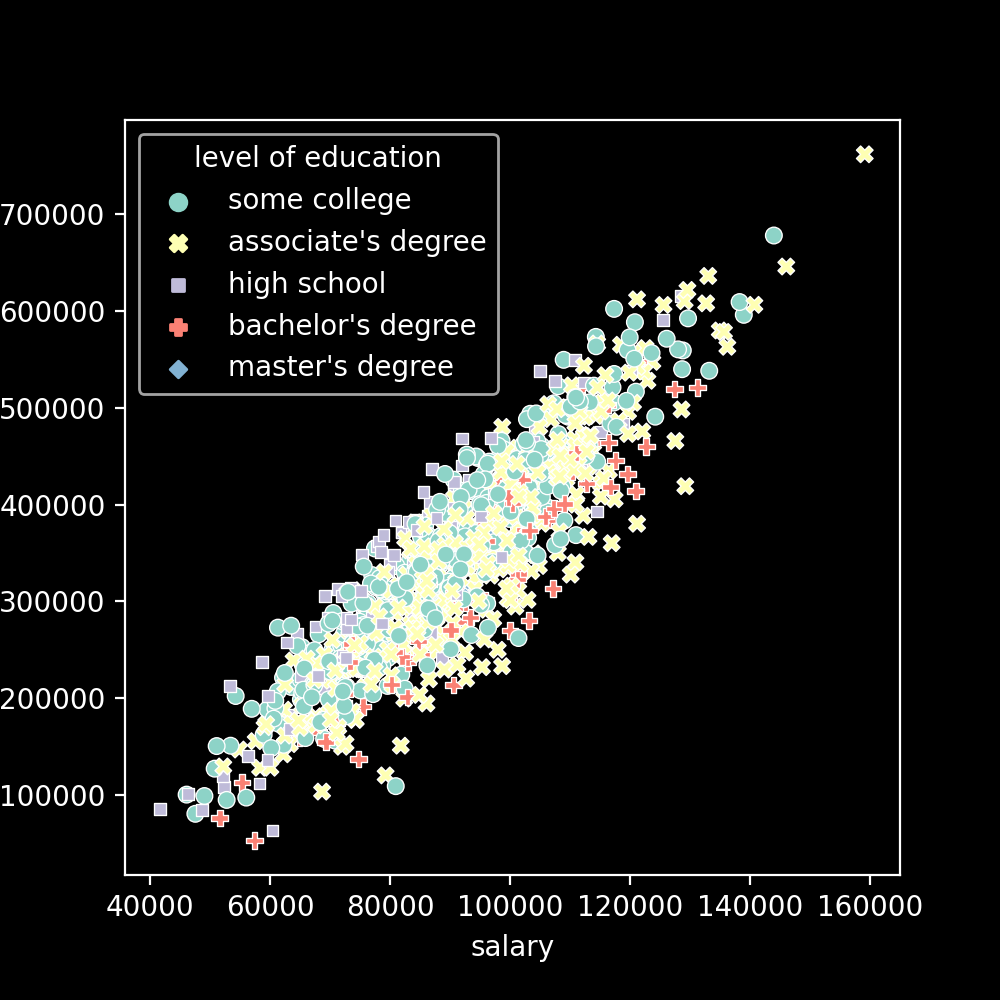
plt.figure(figsize=(5,5),dpi=200)

sns.scatterplot(x='salary',y='sales',data=df,size='salary',palette='Dark2',alpha=0.5)

* alpha parameter is for the transparency of the data points.
* Finally the style(different symbols for different column values) can also be used. Particularly useful when black and white plots are being used.

plt.figure(figsize=(5,5),dpi=200)

sns.scatterplot(x='salary',y='sales',data=df,style='level of education',hue='level of education',alpha=1)



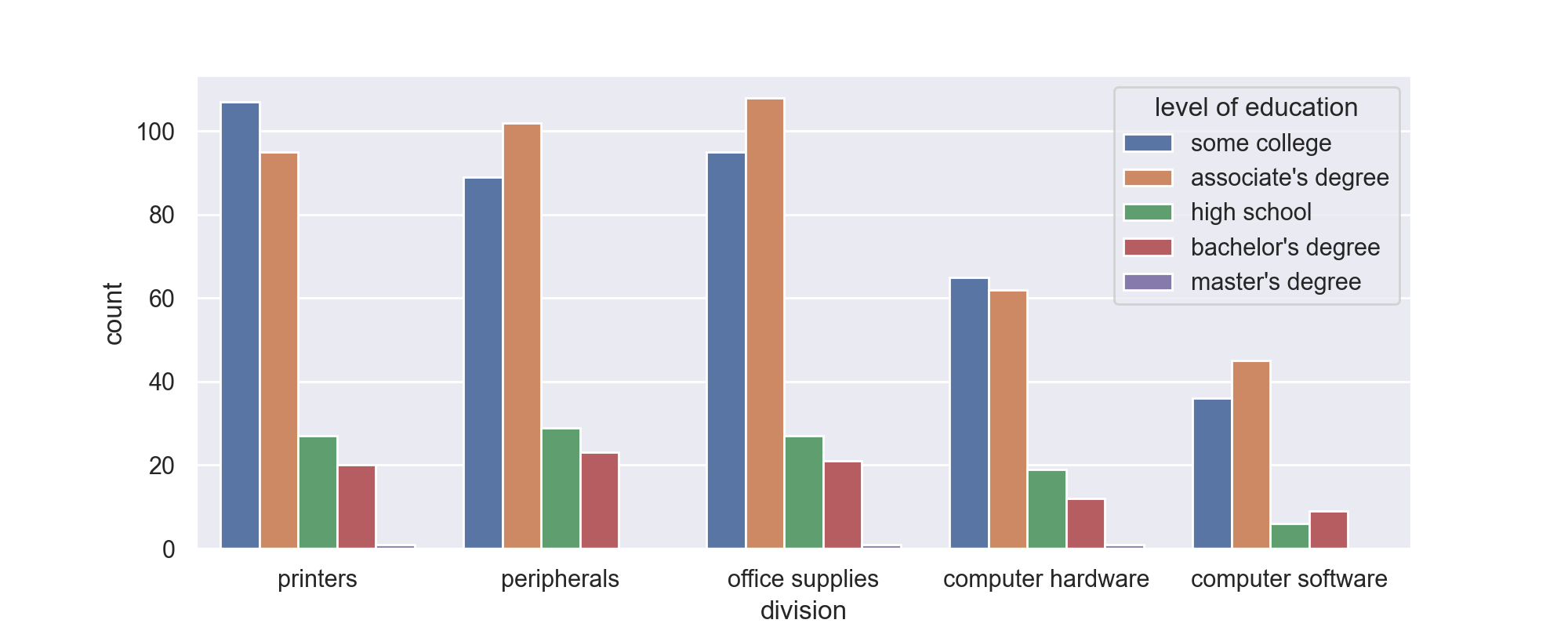
**Categorical Plots**

**Countplot vs bar plot**

* Countplot is used to represent the frequency of occurrence for categorical variables.
* Bar plot is a bit more general. We can represent values like average values of a category as well as their standard deviation.

plt.figure(figsize=(10,4),dpi=200)

sns.countplot(data=df,x='division',hue='level of education')



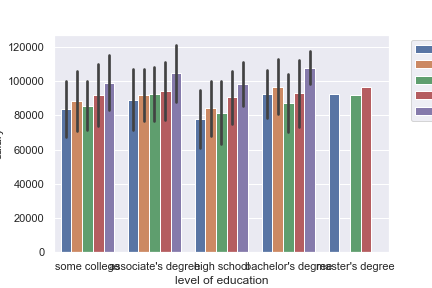
Barplot

#Barplot

sns.barplot(data=df,x='level of education',y='salary',estimator=np.mean,ci='sd',hue='division')

plt.legend(bbox\_to\_anchor=(1.05,1))

plt.savefig('barplot.png')



* As mentioned above, barplot can be used not just to count the occurrence of a categorical variable but also other parameters. Here **estimator** parameter is used to achieve that and the mean is being displayed.
* **‘ci’** parameter stands for confidence interval(the line in each bar). In this case it is being used to depict the standard deviation of the data.
* **bbox\_to\_anchor** is used to position the legend in the plot. Again, the coordinates are relative to the underlying canvas.

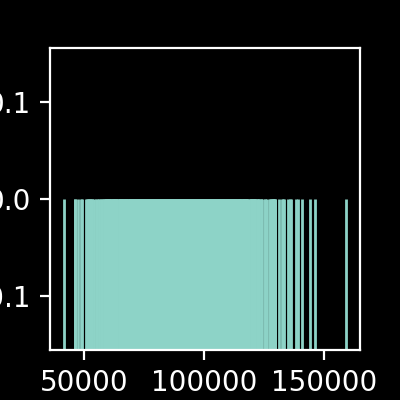
**Distribution plots**

**Rug plot**

* One tick per data point on y axis. Y-axis is meaningless.
* Helps identify outliers

plt.figure(figsize=(2,2),dpi=200)

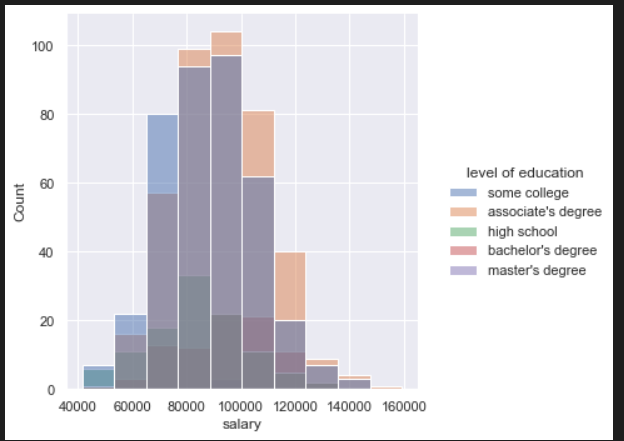
sns.rugplot(x='salary',data=df,height=0.5)



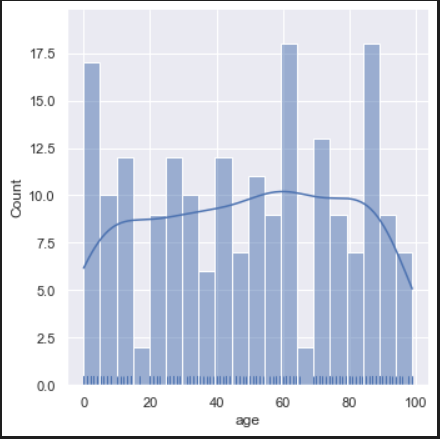
**KDE plot(kernel density estimation plot)**

* Estimating a continuous distribution for a given datapoint.
* For histograms, although there is a **sns.histplot()** available we go for the more general **sns.displot().**

sns.displot(data=df,x='salary',bins=10,hue='level of education')



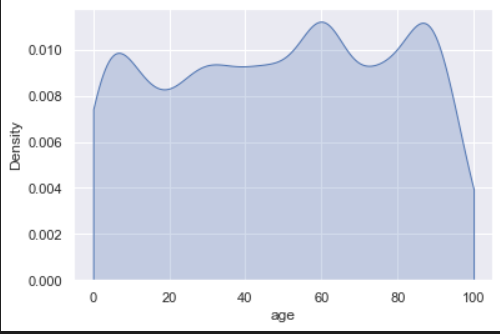
sns.displot(data=sample\_ages,x='age',bins=20,rug=True,kde=True)



* **rug=True and kde=True** parameters can be used to have the kde estimation and Rug plot included within the histogram. Can vary bin size to obtain varying plots.

sns.kdeplot(data=sample\_ages,x='age',clip=[0,100],bw\_adjust=0.6,shade=True)

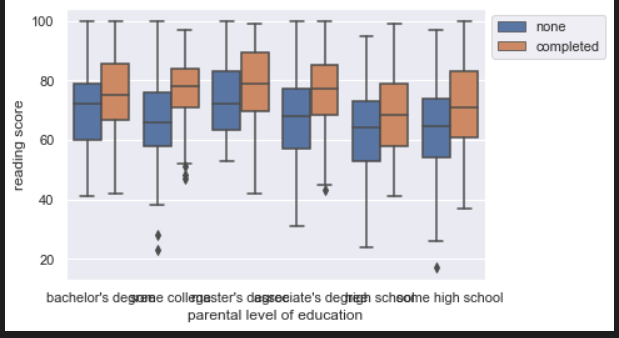
* **Clipping** off values is similar to xlim and ylim. It is to eliminate impossible values like negative salary values.
* Lower the value of **bw\_adjust**, more the variance that is picked up. A high value would mean a high degrees of underfitting
* **shade** fills up the area under the curve



**Distribution Within categories**

**Box plots**

sns.boxplot(data=df,y='reading score',x='parental level of education',hue='test preparation course')



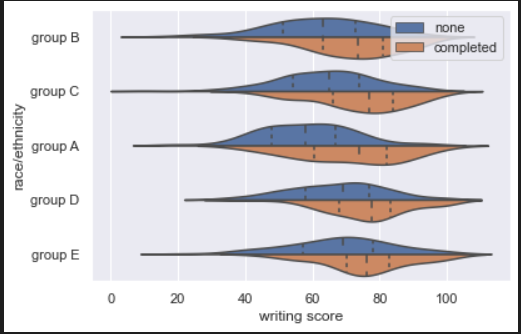
* This can be flipped too depending on which axis the continuous variable is located on. Setting hue to a column value, we are obtaining separate box plots for each column value.

**Violin Plot**

* This is a KDE plot which is mirrored and attached at the bottom.

sns.violinplot(data=df,x='writing score',y='race/ethnicity',hue='test preparation course',split=True,inner='quartile')

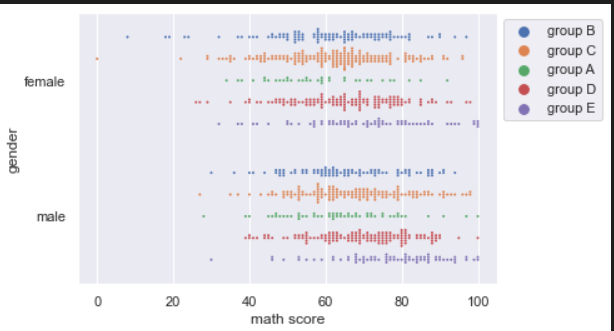
* By having **split=True** one half of violin plot is one category other half is another. This prevents separate violin plots for column plots like in the box plot displayed above.
* Inner = quartile draws lines at each quartile



**Swarm Plot**

#Swarm plot

sns.swarmplot(data=df,x='math score',y='gender',hue='race/ethnicity',size=2,dodge=True)



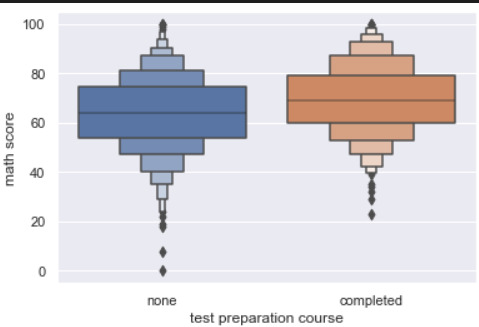
* Hue paramater will be difficult to interpret since we are considering individual data points

. Hence we go for **dodge=true** to split the plot based on the hue parameter

**Boxen Plot**

#Boxen plot: Said to be more informative than normal box plot

sns.boxenplot(data=df,x='test preparation course',y='math score')



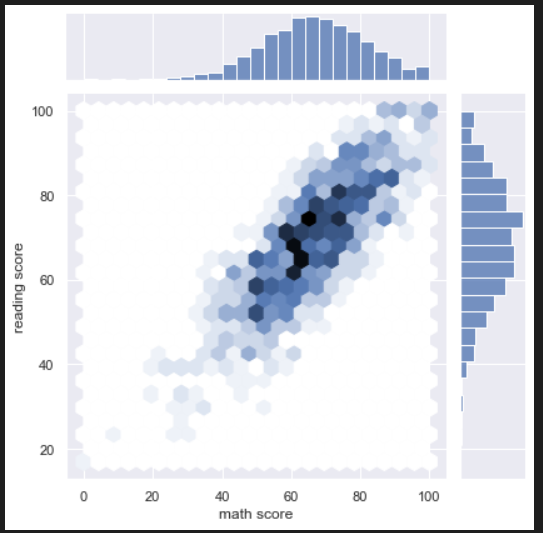
**Comparison plots**

**Joint Plot**

* In this plot when we consider two features, we plot the relation between the two categories(hex/scatter plot) along with the underlying distributions of those features(kde /histogram plot)

sns.jointplot(x='math score',y="reading score",data=df,kind='hex')

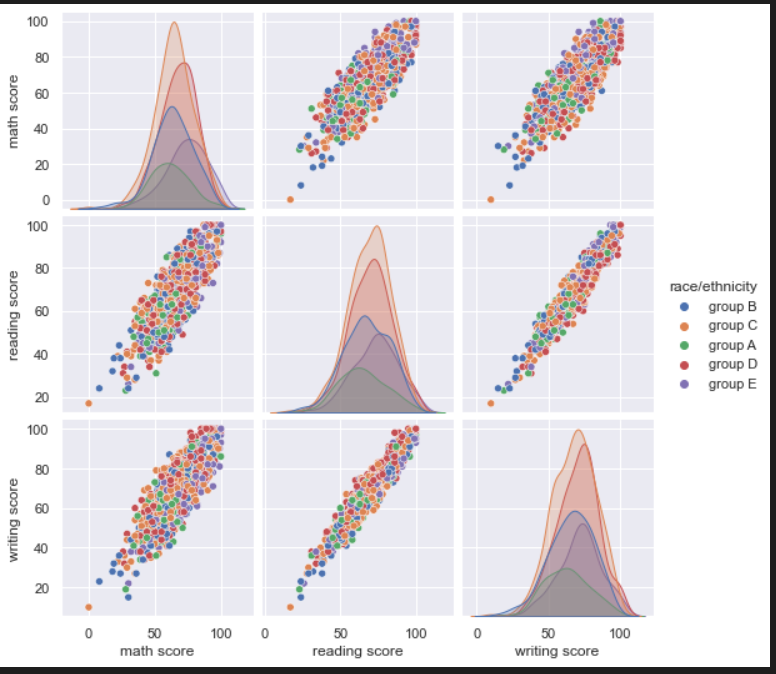
* the kind parameter can be used to specify between plot types be it for the relation plots(hex/scatter) or the underlying distributions(histogram,kde)



**Pairplots**

* It is CPU intensive
* Plots out histograms as well as scatter plots for every possible feature combination
* With pairplots, it is advisable to feed in only the columns we are interested in.

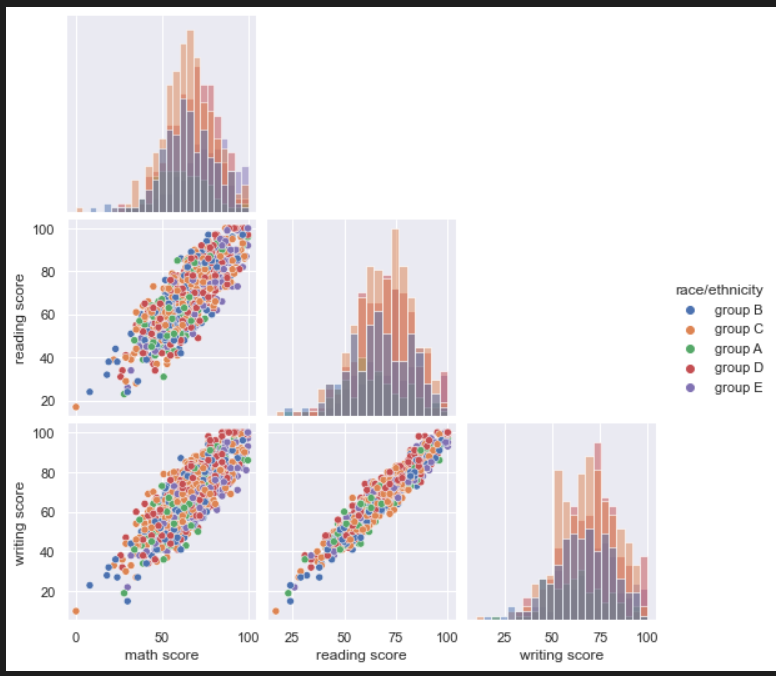
sns.pairplot(data=df,hue='race/ethnicity')



* It can be noticed that the diagonal elements are underlying distributions while the non diagonal elements are relation plots. It is also noticed that the plot is symmetric along the diagonal.
* We have parameters for both of this. That is to get rid of duplicates as well as choose what the underlying distribution type must be (histogram/ kde)

sns.pairplot(data=df,hue='race/ethnicity',diag\_kind='hist',corner=True)

* **corner=True** eliminates the duplicates
* **diag\_kind** parameter is used to ensure that the diagonal elements are histograms only



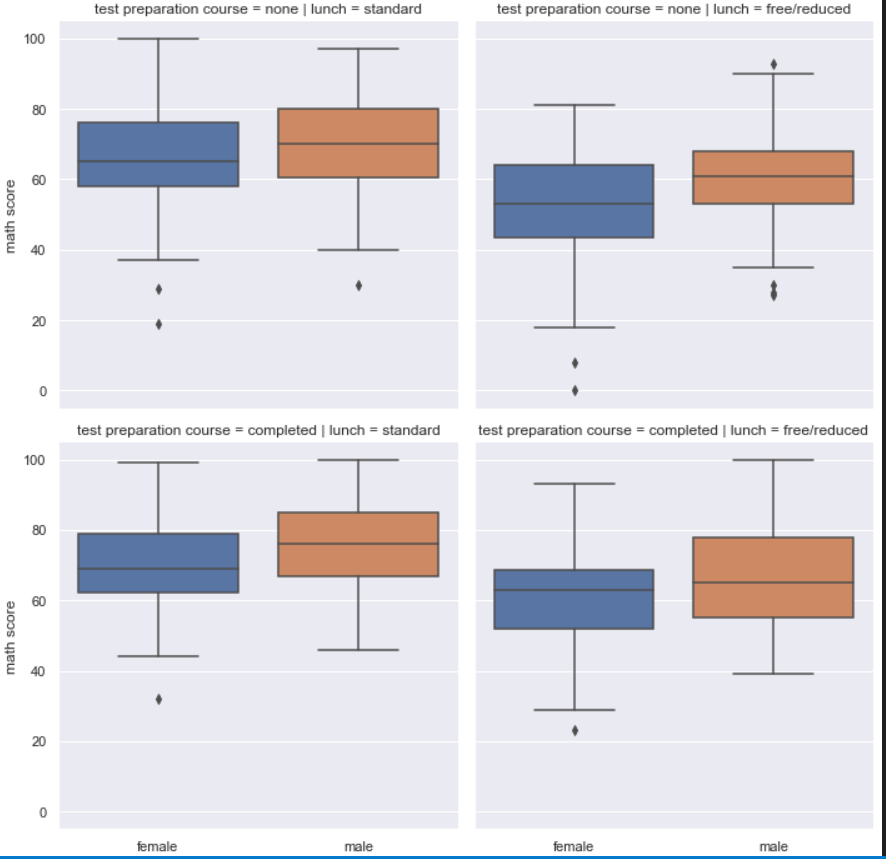
**Grid plots**

* These are equivalent to using the subplot functionality in matplotlib. There are two methods, catplot and pairgrid.

**Catplot**

sns.catplot(data=df,x='gender',y='math score',kind='box',col='lunch',row='test preparation course')

* Advantage to matplotlib is that we don’t need to specify the number of rows and columns. Instead the row and col parameters take in values and the fed features will vary along the rows and columns respectively. The **kind** parameter again is used to specify what kind of plots we need.



**Pairgrid**

* When we use pairplots , internally the pairgrid functionality is called and used.

grid = sns.PairGrid(df,hue='gender')

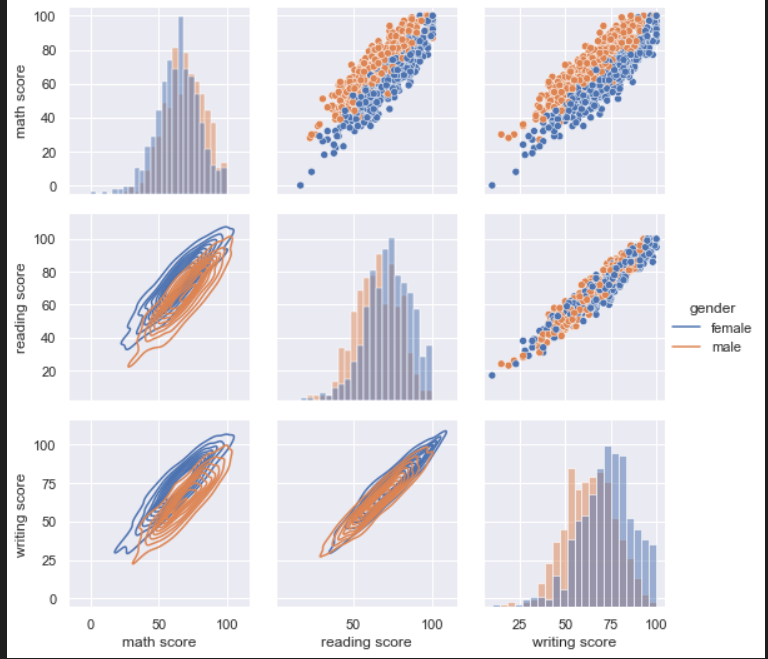
grid=grid.map\_upper(sns.scatterplot)

grid= grid.map\_lower(sns.kdeplot)

grid = grid.map\_diag(sns.histplot)

grid=grid.add\_legend()

* The listed functions are used to customize the plots above, below and along the diagonal respectively. And since grids aren’t added automatically , we need to use the command mentioned in the end.



**Matrix Plots**

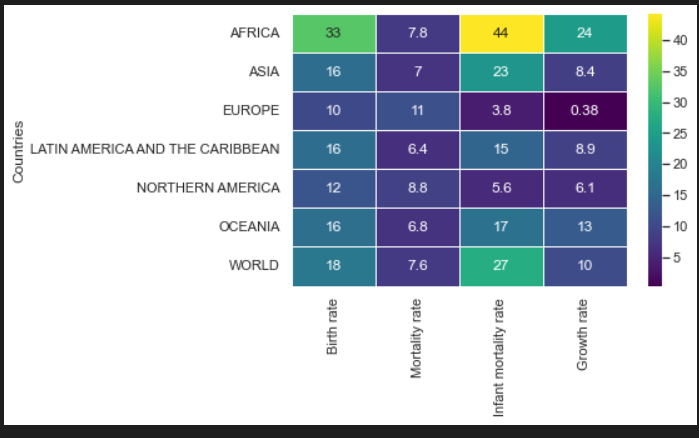
* The purpose of heatmap or cluster-map is to better visualize the volume of locations/events within a dataset and assist in directing viewers towards areas on data visualizations that matter most.
* Ideally all the values should be in the same unit
* Helps identify regions/people with similar features.
* Seaborn can innately group together similar groups

df=df.set\_index("Countries")

* This is a crucial step because it allows us to obtain a more meaningful graphical representation.

sns.heatmap(df.drop('Life expectancy',axis=1),linewidth=0.5,annot=True,cmap='viridis')

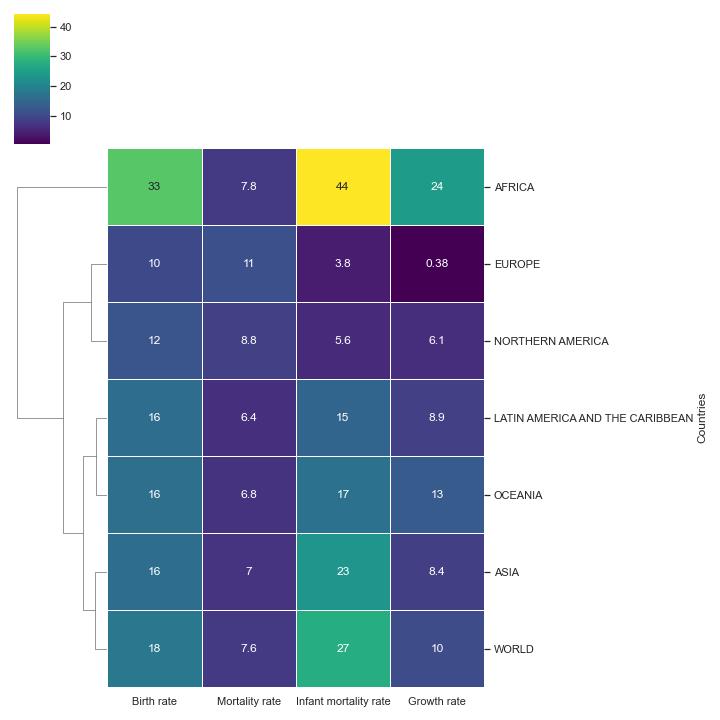
* The life expectancy column in being dropped because it’s units/magnitude is not on scale with the other features.(Others were all rates).
* **Annot= true** to have the value displayed and **linewidth** to better segregate the categories.



**Clustermap**

* This helps us identify the categories with similar values.

sns.clustermap(df.drop('Life expectancy',axis=1),linewidth=0.5,annot=True,cmap='viridis',col\_cluster=False)



**Data preprocessing**

First step will be to extract the input and output feature vectors from the dataset

df = pd.read\_csv("Data.csv")

#input feature vector

x = df.iloc[:,:-1].values

#output feature vector

y = df.iloc[:,-1].values

**Taking care of missing data**

from sklearn.impute import SimpleImputer

imputer = SimpleImputer(missing\_values=np.nan,strategy='mean')

imputer.fit(X[:,1:3])

X[:,1:3] = imputer.transform(X[:,1:3])

* Note that unlike in other methods, fit and transform are two separate methods.
* The SimpleImputer function takes in two arguments:

**missing\_values :** what values must be considered as missing data (np.nan in this case)

**strategy :** what do we replaced the missing data with. Mean in this case

* **Only the numerical columns** must be fed to the fit and transform methods.

**Encoding categorical data**

* Whenever we have categorical data in our data, we need to encode it using dummy variables. This can be achieved using two methods:
  + Label Encoder
  + One Hot Encoder

**Input features**

from sklearn.compose import ColumnTransformer

from sklearn.preprocessing import OneHotEncoder

ct=ColumnTransformer(transformers=[('encoder',OneHotEncoder(),[0])],remainder='passthrough')

X = np.array(ct.fit\_transform(X))

* ColumnTransformer function takes in 2 arguments:

**transformers :** In this parenthesis we specify the action(encoder), the specific function being used (OneHotEncoder()) and the index of the column on which the action must be performed([0])

**remainder = ‘passthrough’:** This argument is added so that the other columns(ones on which the transform hasn’t being applied are returned too.

* The output of the fit\_transform function has to be converted to a np.array because that is the format in which it will be fed to various machine learning algorithms.

**Output Features**

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

y = le.fit\_transform(y)

* In this case it is not necessary for the output to be in np.array format

**Test-Train Split**

from sklearn.model\_selection import train\_test\_split

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

* **test\_size :** Proportion of data that must be set aside as test data.
* **Random\_state :** set seed so that the results can be comparable.

**Feature scaling**

* We do this to avoid some features dominating others(this can happen due to difference in magnitudes)
* Feature scaling must be done after test train split. Feature scaling needs to be applied only for train set. We then use the coefficients/ scalar obtained from this to transform the test data.
* Standardization vs normalization: Standardization works all the time. Normalization recommended when the features are distributed normally
* We should be careful not to apply feature scaling to the dummy variables. (encoded ones)

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X\_train[:,3:] = sc.fit\_transform(X\_train[:,3:])

X\_test[:,3:] = sc.transform(X\_test[:,3:])

* Choose the numerical , non categorical columns from X\_train. ‘fit’ will obtain values like mean and standard deviation while transform will use them to standardize the data.
* For test data, we **use only ‘transform’. T**his is because the scalars have already been calculated via train data and the same must be reused for test data.

**Simple Linear Regression**

Assumptions of linear regression:

* Linearity(relation between the dependent and independent variables must be linear. Can be checked via scatter plots)
* Homoscedasticity : Variance with respect to the residuals must be equal/constant throughout the entire range of the independent variable. I.e, the residuals must be equal along the regression line
* Multivariate Normality(can be checked via histogram or a QQ plot)
* Independence of errors(No auto-correlation)
* Lack of multi-collinearity(This occurs when the coefficients are highly correlated to one another . This can be tested by observing the values in the correlation matrix for Pearson coefficients. The values must be less than 1).

from sklearn.model\_selection import train\_test\_split

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

* Splitting the data into test and train sets.

**Building the model**

from sklearn.linear\_model import LinearRegression

regressor = LinearRegression()

regressor.fit(X\_train,y\_train)

* Feed the X\_train and y\_train vectors to the fit function.

**Making predictions**

y\_pred = regressor.predict(X\_test)

* Using the model already trained in the previous set, we obtain the predictions for our test data.

**Visualizing the results**

*For Training set*

plt.scatter(X\_train,y\_train,color='red')

plt.plot(X\_train,regressor.predict(X\_train))

plt.title('Salary vs Experience(Training Set)')

plt.xlabel('Years of Experience')

plt.ylabel('Salary')

plt.show()

For Test Set

plt.scatter(X\_test,y\_test,color='red')

plt.plot(X\_train,regressor.predict(X\_train))

plt.title('Salary vs Experience(Test Set)')

plt.xlabel('Years of Experience')

plt.ylabel('Salary')

plt.show()

* Note : While visualizing the results for the test set, notice that we are using X\_train instead of X\_Test. This makes no difference since the model has already been trained using the training set data and now we are merely making predictions for the new data using the existing model.

**Getting model Predictions for a single value**

regressor.predict([[value]])

* Using this we can obtain the value for individual data points.

**Obtaining the model coefficients**

print(regressor.coef\_)

print(regressor.intercept\_)

y = **(regressor.coef\_)\* X + (regressor.intercept\_)**

**Multiple Linear regression**

* **Dummy Variable Trap:** When we use One hot encoder to encode categorical variables, the resultant dummy variables are highly correlated which makes it difficult for the model to interpret the regression models.
* Example a column has values ‘male’ and ‘female’. Use one Hot encoding we get (1,0) for male and (0,1) for female. Let the column be [male,female,male,male,female,male] . So D1(male) = (1,0,1,1,0,1) and D2(female) – (0,1,0,0,1,0). It can be observed that if D1= 1 then D2=0 and vice versa. i.e, D1 = 1-D2. In this case while training the model, we can drop one of these redundant features.
* This applies when there are more than two categories also. **We should always omit one dummy variable.**

**P-Code and statistical significance**

* Lower value of p-value means that the Null hypothesis has been disproved and the alternate hypothesis holds. This means that the effect of the feature being considered is statistically significant

**Model Building**

5 methods of model building

* **All – in** : Throw in all the variables. We may adopt this approach when we have prior knowledge ,are forced to use all the features or if we are preparing for backward elimination method.
* **Backward elimination(fastest method)**:
  + Select a significant level to stay in the model
  + Fit the model with all possible predictors.
  + Consider the predictor with the highest P-value. If p-value >SL, go to next step or go to FIN
  + Remove the predictor
  + Fit the model without this variable.
* **Forward Selection:**
  + In this we start with just the intercept and keep on adding the variables as long as our model performance keeps improving
  + Select a significance level to enter the model
  + Fit all simple regression models y~xn. Select the one with lowest p-value.(Like one variable at a time)
  + Keep this variable and fit all possible models with one extra predictor added to the one you already have
  + Consider the predictor with the lowest P-value. If P<SL go to previous step. Otherwise go to FIN.
* **Bidirectional Elimination:**
  + Select a significance level to enter and to stay in the model
  + Perform the next step of forward selection(new variables must have P<SLENTER to enter)
  + Perform all steps of Backward elimination(old variables must have P<SLSTAY to stay)
  + Ultimately no new variables can enter and no old variables can exit.
* **All possible models(Score comparison):**
  + Select a criterion of goodness of fit
  + Construct all possible regression models : 2^N -1 total combinations
  + Select the one with the best criterion
  + This can get highly tedious as the number of input feature columns rises.
* The multiple linear regression class in sklearn will take care of the dummy variable trap. Even the backward elimination method is implemented by the sklearn library.

**Training the model**

from sklearn.linear\_model import LinearRegression

regressor = LinearRegression()

regressor.fit(X\_train,y\_train)

**Bias-Variance Trade-Off**

* Also known as underfitting (high bias) vs overfitting (high variance). Our model should generalize well to new unseen data but should also account for variance and patterns in the known data.
* Overfitting:
  + Hard to catch because of low error on training sets. Issue comes up when used on test sets.
* Underfitting:
  + Poor performance even on training set. Generalizing too much and high bias
* In an ideal world, as the model complexity increases, the error should decrease. The strategy for choosing the ideal model complexity of the model remains the same. Train the model on the training set, make predictions on the test set and then plot out/compare the errors. That is we must consider both train error and test error while choosing model complexity.

**Feature Engineering and Data preparation**

* **Integer Encoding:**
* Easy to understand and does not increase the number of features.
* But it can imply ordered relationship between categories
* **One hot Encoding (dummy variables):**
* This can however expand the number of feature columns
* Dummy variable trap needs to be considered

**Polynomial regression**

Helps overcome two shortcomings of linear regression

* Non-linear relationship between independent and dependent variables
* To account for the interaction between the features themselves. This basically takes care of synergy.
* Basically, these are features which might sound insignificant when considered individually but the add great value when considered in combination. We multiply these terms to obtain the interaction terms. In Scikit learn we have the **PolynomialFeatures** function to take care of this.
* Approach is to build a matrix of features with the polynomial features and then integrate it into the linear regression model.

**Creating polynomial features**

* The key step here is to a build a matrix containing the polynomial powers of the input feature.

from sklearn.preprocessing import PolynomialFeatures

poly\_feat = PolynomialFeatures(degree = 4)

X\_poly = poly\_feat.fit\_transform(X)

* We can increase the degree of the polynomial to obtain a better prediction for our input data. But we must be careful since this might lead to overfitting.

**Training the model**

lin\_reg2 = LinearRegression()

lin\_reg2.fit(X\_poly,y)

* So instead of feeding the independent feature directly into the linear regression model, we feed the polynomial features instead.

**Visualizing the results**

plt.scatter(X,y,color='red')

plt.plot(X,lin\_reg2.predict(X\_poly),color='blue')

plt.title('Truth or Bluff(Polynomial Regression)')

plt.xlabel('Position Level')

plt.ylabel('Salary')

plt.show()

**Obtaining predictions for new data**

* For linear regression, to obtain prediction for a value ‘val’ we would use:

regressor.predict([[val]])

* Since we are feeding a polynomial feature array in this case , we can obtain the predictions by the following:

pol\_build = PolynomialFeatures(degree=4)

lin\_reg2.predict(pol\_build.fit\_transform([[6.5]]))

**Selecting the optimal degree of the polynomial**

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error

train\_rmse\_errors = []

test\_rmse\_errors = []

for d in range(1,10):

    poly\_converter=PolynomialFeatures(degree=d,include\_bias = False)

    poly\_features = poly\_converter.fit\_transform(X)

    X\_train,X\_test,y\_train,y\_test = train\_test\_split(poly\_features,y,test\_size=0.2)

    model = LinearRegression()

    model.fit(X\_train,y\_train)

    train\_pred = model.predict(X\_train)

    test\_pred = model.predict(X\_test)

    train\_rmse = np.sqrt(mean\_squared\_error(y\_train,train\_pred))

    test\_rmse = np.sqrt(mean\_squared\_error(y\_test,test\_pred))

    train\_rmse\_errors.append(train\_rmse)

    test\_rmse\_errors.append(test\_rmse)

* This is the exact same procedure as training a polynomial regression model. Here we are putting that in a loop and appending error values for different degrees of polynomial.
* train\_rmse and test\_rmse have the errors for the test and train set respectively. So at whatever degree we observe we observe a sharp increase in error it means there is overfitting and we need to decide the degree of the polynomial accordingly.

**Visualising the errors**

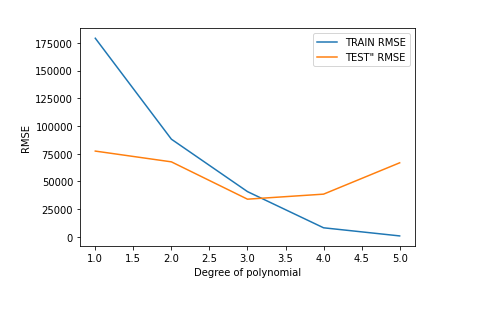
plt.plot(range(1,6),train\_rmse\_errors[:5],label ='TRAIN RMSE')

plt.plot(range(1,6),test\_rmse\_errors[:5],label ='TEST" RMSE')

plt.ylabel('RMSE')

plt.xlabel('Degree of polynomial')

plt.legend()



* In the above graph, we can observe the error increase after degree 3. So, it optimal to choose 2 as the degree of polynomial.

**Grid Search**

* It is a way of training and validating a model using every possible value of hyperparameter options

**Step 1: Initializing the model**

from sklearn.model\_selection import train\_test\_split

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

from sklearn.linear\_model import ElasticNet

base\_elastic\_net\_model = ElasticNet()

* First we perform the train-test split and initialize the model we want to use. In this case , the ElasticNet model is being used.

**Step 2: Creating a parameter grid for the hyperparameters**

from sklearn.model\_selection import GridSearchCV

param\_grid = {'alpha':[0.1,1.5,10,50,100],'l1\_ratio':[.1,.5,.7,.95,.99,1]}

grid\_model =GridSearchCV(estimator = base\_elastic\_net\_model,param\_grid=param\_grid,scoring='neg\_mean\_squared\_error',cv=5,verbose =0)

grid\_model.fit(X\_train,y\_train)

* The ElasticNet model used has two important hyperparameters, **alpha and l1\_ratio**. SO we define a **param\_grid** which is basically a dictionary with the hyperparameter name as the key value and the list of possible values for that hyperparameter as a value.
* Then we import the **GridSearchCV** module.
* For the **estimator** we feed in the model we are using (which is elasticnet in this case).
* **Parameter grid** we created is fed in next. **Scoring** parameter has predefined values. It is basically telling the model which metric to consider while choosing the ideal hyperparameter value. In this case we are going for **neg\_mean\_squared\_error**(least value of mean squared error)
* The **cv** parameter determines how many folds cross validation we have to perform and **verbose** determines how many results need to be displayed.

**Obtaining the best parameters and predictions**

grid\_model.best\_estimator\_ #returns the values

grid\_model.best\_params\_ #returns the same but as a dictionary

y\_pred = grid\_model.predict(X\_test)

* **While making predictions the model choose the best approximation parameters.**

**Regularization**

* **L1 regularization:** Adds a penalty equal to the absolute value of the magnitude of coefficients
* **L2 regularization:** Adds a penalty equal to square of magnitude of coefficients
* **Elastic net :** Combines L1 and L2 with the addition of an alpha parameter

**Feature scaling:**

* + Improves the convergence of steepest descent algorithms(example gradient descent)
  + Allows us to compare model coefficients of one model to another (since the different units will be taken care of)

*Two methods of achieving this:*

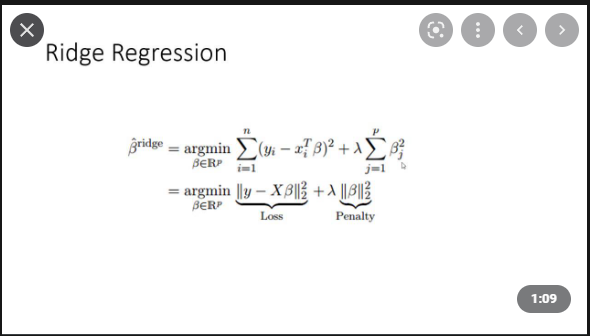
* + **Standardization:** Rescales data to have a mean of 0 and SD of 1. Also called as Z-score normalization. The data points are fit to a standard normal distribution
  + **Normalization :** Rescales all data values to be between 0-1

*Feature scaling in Scikit learn:*

* A.fit() – only calculates the necessary statistics9min, max, mean and SD)
* A.transform() – scales the data and returns the transformed version of data
* **We only fit to training data**. Further scaling of new data will be based on the parameters obtained from this.(i.e only use transform on new data/test data)

**Regularization**

**Ridge regression:**



* Lambda will determine how severe the penalty will be.
* Intuitively we can notice that the beta parameters are slopes of the regression line. The squared term will **punish higher slopes (higher coefficient values**). So, introduction of the ridge regression term will result in **less steep regression lines which reduces over fitting.**
* **Trying to minimize a squared beta term leads us to punish larger coefficients.**

**Code**

Let us start out by not tuning the hyper parameter alpha (which is lambda in the image above).

*Initialising and training the model*

from sklearn.linear\_model import Ridge

ridge\_model = Ridge(alpha=10)

ridge\_model.fit(scaled\_x\_train,y\_train)

**Predictions and obtaining the performance metrics**

test\_predictions = ridge\_model.predict(scaled\_x\_test)

from sklearn.metrics import mean\_absolute\_error,mean\_squared\_error

MAE = mean\_absolute\_error(y\_test,test\_predictions)

RMSE = np.sqrt(mean\_squared\_error(y\_test,test\_predictions))

**Tuning the hyperparameter(optimal value of alpha)**

from sklearn.linear\_model import RidgeCV

ridge\_cv\_model = RidgeCV(alphas=(0.1, 1, 10),scoring='neg\_mean\_absolute\_error')

ridge\_cv\_model.fit(scaled\_x\_train,y\_train)

* Instead of Ridge, we use RidgeCV. This function performs cross validation for a variety of alpha values and reports back the best alpha value

ridge\_cv\_model = RidgeCV(alphas=(0.1, 1, 10),scoring='neg\_mean\_absolute\_error')

ridge\_cv\_model.fit(scaled\_x\_train,y\_train)

**To obtain the optimal alpha value:**

ridge\_cv\_model.alpha\_

We can feed a tuple of alpha values**. Scoring** parameter determines the way in which the optimal alpha value is determined. To obtain the list of possible scoring techniques , we can use the following commands:

from sklearn.metrics import SCORERS

SCORERS.keys()

**Predictions and obtaining the performance metrics**

We notice that the performance would have improved when the hyper parameters are tuned.

y\_pred = ridge\_cv\_model.predict(scaled\_x\_test)

MAE = mean\_absolute\_error(y\_test,y\_pred)

RMSE = np.sqrt(mean\_squared\_error(y\_test,y\_pred))

It is noticed that none of the model coefficients are zero (no features have been deemed redundant or totally eliminated). To obtain the model coefficients:

ridge\_cv\_model.coef\_

**Lasso regression(L1 regularization)**

* Instead of the squared coefficient penalty term as in ridge, here we use the sum of absolute value of coefficients.
* Unlike the ridge model, this can lead to sparse models which means it can force some of the coefficient estimates to be exactly equal to zero.(they are deemed redundant or insignificant)

from sklearn.linear\_model import LassoCV

lasso\_model = LassoCV(eps=0.001,n\_alphas=100,cv=5,max\_iter=1000000)

* We can give list of alphas like in ridgecv. But there is an option to give epsilon and n\_alphas.
* **epsilon** = alpha \_min/alpha\_max, n\_alphas = number of alpha values desired. Lower the value of epsilon, broader the range of values checked
* We can choose a higher value of **max\_iter** if the result won't converge

lasso\_model.fit(scaled\_x\_train,y\_train)

lasso\_model.alpha\_ #Obtain the optimal alpha value in the given range

**Predictions and obtaining the performance metrics**

y\_pred = lasso\_model.predict(scaled\_x\_test)

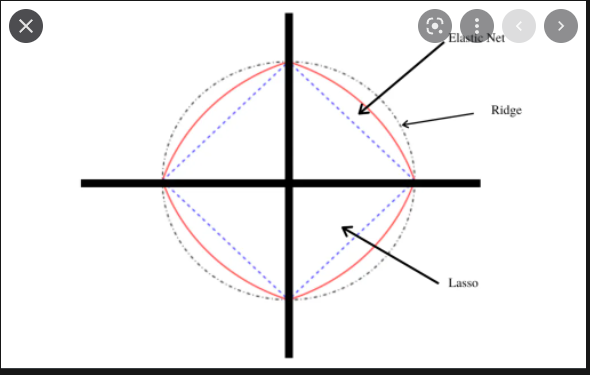
MAE = mean\_absolute\_error(y\_test,y\_pred)

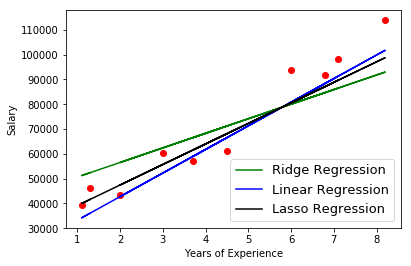
RMSE = np.sqrt(mean\_squared\_error(y\_test,y\_pred))

* Although the performance metrics are better in ridge than lasso, the importance of lasso is underscored when we check the coefficients. It is observed that most of them have become zero, only some of the features have been used in the model.
* **Basically lasso model is simpler meanwhile ridge model has better performance metrics.**
* Model performance can be enhanced by choosing a smaller epsilon value (broader range) and allowing more iterations.

**Elastic Net regression(combining both L1 and L2 regularization)**

* Both lasso and ridge regression are subject to condition that the sum of absolute values and the squared terms are less than some sum ‘**s’** respectively.
* When plotted, it is observed that the constraints for lasso form a hypercube (a square in 2 dimensions) while it is a circle for ridge.
* While trying to arrive at the optimal value, the optimal value line/curve will be tangential in case of ridge while it may be along one of the corners of the hypercube for lasso. This is what leads to some of the coefficients going to zero. (need to understand this better)
* Elasticnet seeks to improve on both l1 and l2 by combining them.





**Code**

from sklearn.linear\_model import ElasticNetCV

elastic\_model = ElasticNetCV(l1\_ratio=[.1, .5, .7, .9, .95, .99, 1],eps=0.001,n\_alphas=100,max\_iter=10000)

* The eps and n\_alphas parameters are same as in lasso regression. **l1\_ratio**  lets us determine whether l1 or l2 needs to be used. Ideally it should lean more towards l1 regression(simpler model)

elastic\_model.fit(scaled\_x\_train,y\_train)

elastic\_model.l1\_ratio\_

* This helps us determine which regularization would best suit our data. So instead of lasso or ridge individually, it is advisable to go for elasticnet and obtain the suitable l1\_ratio.

**Cross Validation**

**Support Vector Machines**

* Based on a simple intuition to check if a hyperplane exists that can effectively separate different classes. New points will fall on one side of this hyperplane and we can assign it to a class.
* In a N dimensional space, a hyperplane is a flat affine subspace of dimension N-1

1-D : point

2-D : line

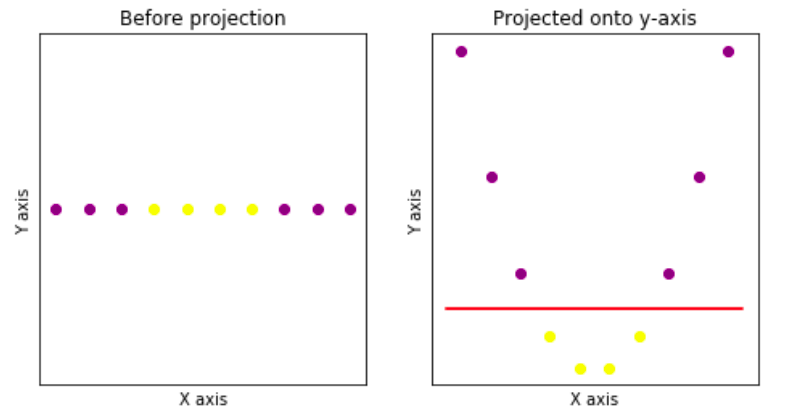
3-D : flat plane

**Soft margins, Maximum Margin Classifier and Support Vector Classifier**

* We choose the separator that maximizes the margins between the classes. This is called the **Maximal Margin Classifier.**
* When the classes are not perfectly separable, we allow for a bias-variance tradeoff. The distance between the threshold and the observations is a **soft margin.** This soft margin allows for misclassification within the margin.
* We can use cross -validation to figure out the best threshold value. (Procedure similar to how we determined the degree of polynomial in polynomial regression)
* This is where the difference between maximum margin classifier and Support vector classifier comes. **Support vector classifier allows for soft margins (**i.e., using bias-variance tradeoff it allows for classification errors in order to achieve better results on future data).
* When we cannot fit a hyperplane even while allowing generous soft margins, we move from support vector classifier to Support vector machines.

**Kernels**

* Kernels are used to project our existing features onto another dimension.

****

* As observed in the plot above, we use kernels on points which are inseparable using a linear hyperplane, project them onto a higher dimension and then find a plane/ basis for separation.

**Mathematical constraints for maximal margin classifier:**

* **(X)=β0+β1X1+β2X2+….. βNXN**: This is the equation of the hyperplane. We need to maximise the margin M with respect to these beta – parameters.
* **yi(x′iβ+β0)≥M**: This constraint basically means that there shouldn’t be any points between the margins(no soft margin)

**Mathematical constraints for support vector classifier:**

* The constraints are similar to maximal margin classifier barring a slight addition

**yi(x′iβ+β0) ≥ M(1- e(i)) : e(i)** is the error term which accounts for the soft margin . Basically we allow for a certain number of misclassifications ‘C’ (which is summation over n of e(i)) to get better results in the future.

* Note that as the number of misclassifications allowed **(C)** goes to zero, support vector classifier approaches the maximal margin classifier.

**Support vector machines:**

* We use the kernel trick which uses the dot product to reduce computational complexity.**(Read more about the mathematical reasoning behind this)**
* Dot product can be described as a measure of similarity between two vectors.
* One important thing to be noted. The location of the hyperplane is determined mainly based on the support vectors. Support vectors are data points that lie on the edge of the class / that are closer to the hyperplane and influence its position and orientation.
* So in the kernel trick we calculate the inner products of all pairs of training observations. Since the non-support vectors are inconsequential in determining this, we only consider the indices of the support vectors to perform the dot product.
* A kernel is a function that quantifies the similarity of two observations. Dot product gives us the measure of similarity between two vectors, similarly the kernel acts as a measure of similarity between the original feature space and the enlarged feature space. (**Need to watch a youtube video regarding this.)**

**SVM Classification**

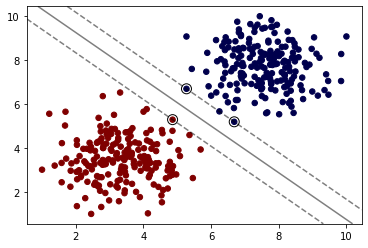
from sklearn.svm import SVC

model = SVC(kernel = 'linear',C = 1000)

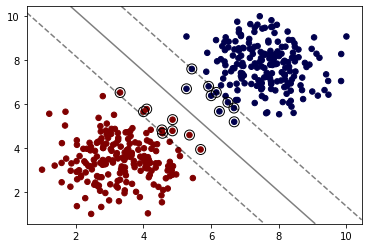
model.fit(X,y)

* The important thing here is choosing the kernel and C value. By default ‘radial basis function’ kernel is used since it usually gives the best results
* C is where the bias variance tradeoff comes in. C is the number of misclassifications we allow (soft margin). The sklearn regularization parameter is inversely proportional to the conventional C value. That is, smaller the value of C we give , the more data points we allow between the margins.

**C=1000**



**C=0.005**

****

**Choosing rbf kernel**

model\_rbf = SVC(kernel='rbf',C=1,gamma='auto')

model\_rbf.fit(X,y)

plot\_svm\_boundary(model\_rbf,X,y)

* Gamma parameters define how far the influence of a single example reaches. **As it gets larger** it assigns more influence to each support vector and might lead to **overfitting.**
* We can either use the default value 'scale' or use cross validation to arrive upon the optimal gamma value.

**Using grid search to choose the optimal value of C and kernel to be used**

from sklearn.model\_selection import GridSearchCV

svm =SVC()

param\_grid = {'C':[0.01,0.1,1],'kernel':['linear','rbf']}

grid = GridSearchCV(svm,param\_grid)

grid.fit(X,y)

* We can use **grid.best\_params\_** to obtain the best value of C and the kernel to be used. We can modify this to choose the optimal value for **gamma** in rbf.

**SVM Regression**

* Finding the correlation between features:

sns.heatmap(df.corr(),annot=True)

* annot = True will print out the correlation values
* For algorithms like support vector machines where the geometric feature spaces play a key role, it is recommended to **scale the features.**

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

scaled\_X\_train = scaler.fit\_transform(X\_train)

scaled\_X\_test = scaler.fit\_transform(X\_test)

* Caution: scaling needs to be done only for the input features.

**Model training:**

from sklearn.svm import SVR

base\_model = SVR()

base\_model.fit(scaled\_X\_train,y\_train)

* LinearSVR can only be used in place of SVR and it is faster too. But it only considers linear kernel.
* **Epsilon** hyperparameter: how much error we are willing to allow for each training example. If this is zero, it means we are overfitting.

**Calculating the error metrics:**

* To check if a given mean squared error or the mean absolute error is okay or not, compare it with the mean value of the data and see how it holds up.

from sklearn.metrics import mean\_absolute\_error,mean\_squared\_error

mean\_absolute\_error(y\_test,base\_preds)

from sklearn.metrics import mean\_absolute\_error,mean\_squared\_error

mean\_absolute\_error(y\_test,base\_preds)

* Since there are many hyperparameters involved, it is better to go for grid search to choose the optimal values for these parameters.

from sklearn.model\_selection import GridSearchCV

param\_grid =

{'kernel':['rbf','linear','poly'],'C':[0.005,0.5,1,10,100],'epsilon':[0,0.001,0.05,0.1,0.5],'degree':[2,3,4],'gamma':['scale','auto']}

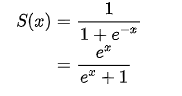
svr\_grid = GridSearchCV(estimator=base\_model,param\_grid=param\_grid,scoring='neg\_mean\_squared\_error',cv=5,verbose=0)

svr\_grid.fit(scaled\_X\_train,y\_train)

**Logistic Regression**

* Allows us to predict categorical data based on historical feature data. For continuous data we create bins (ranges) and then proceed with classification. We can’t use the same error metrics as in regression.

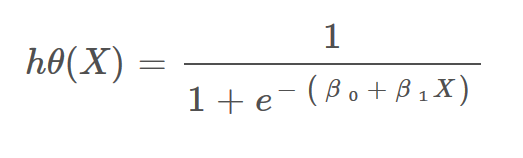
**Logistic/Sigmoid function:**



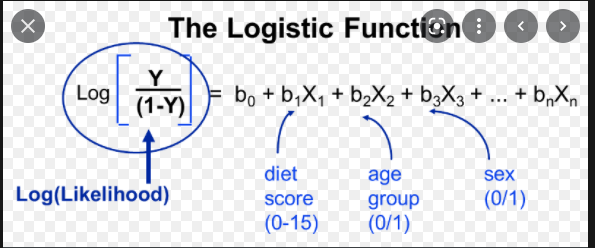
* For any value of x, we have output range between 0 and 1. Many natural real world systems have a **carrying capacity** or a natural limiting factor.
* Treating the y axis as the probability of belonging to a particular class, we set a cut-off probability value. Anything above that will be one class and below that will be another class.

**Linear to logistic intuition:**

* We choose the y-axis as the probability of the input feature belonging to a particular class.



* With the parameter (beta) being in the denominator it becomes difficult to interpret it, hence we take log of this to develop a more nuanced understanding of the topic.
* Odds of event is probability of event happening divided by the chance of it not happening. **(p/1-p).** Solving the above equation for log odds:



X- input feature b – Beta value

* Observation from the plot. When the log odds ratio is considered as the Y-axis, we notice that for **Y =0.5, it is 0 and as Y approaches 1 and 0 the graph goes to infinity and -infinity** respectively. So basically, this gives us separation into classes.
* So in short, in linear regression:

***y = beta coefficients \* input features***

In logistic regression:

***log odds = beta coefficients \* input features***

The log part is crucial in helping us obtain the categorical classification.

**Interpreting Beta Coefficients**

* **Sign of coefficient:** Positive beta value indicates increase in likelihood of belonging to a class with increase in associated input feature(x) **(positive correlation**). Negative beta value indicates decrease in likelihood of belonging to a class with increase in associated input feature(x).**(negative correlation)**
* **Magnitude of coefficient:** Difficult to interpret easily. Comparing the magnitudes of coefficents against each other can lead to insight over which features have the strongest effect on prediction output.

**Best fit with maximum likelihood:**

* P = **elog(odds)/(1+ elog(odds) )**: This can be used to convert the log odds value back to probability values.
* We choose a line in the **log odds axis**. By line we mean the countless prediction lines that can be drawn based on the values on log odds values obtained. We then **project the x(input data points) onto this line.**
* Then, obtain the probabilities using the above formula and then plot them out on the probability axis.
* **Likelihood = Product of probabilities of belonging to class 1.** We need to maximise this so as to make it as close to 1 as possible.
* **Therefore our objective will be to choose the best line from log adds axis whose corresponding probability values maximise the likelihood function.**

**Training the model**

from sklearn.linear\_model import LogisticRegression

logmodel = LogisticRegression()

logmodel.fit(X\_train,y\_train)

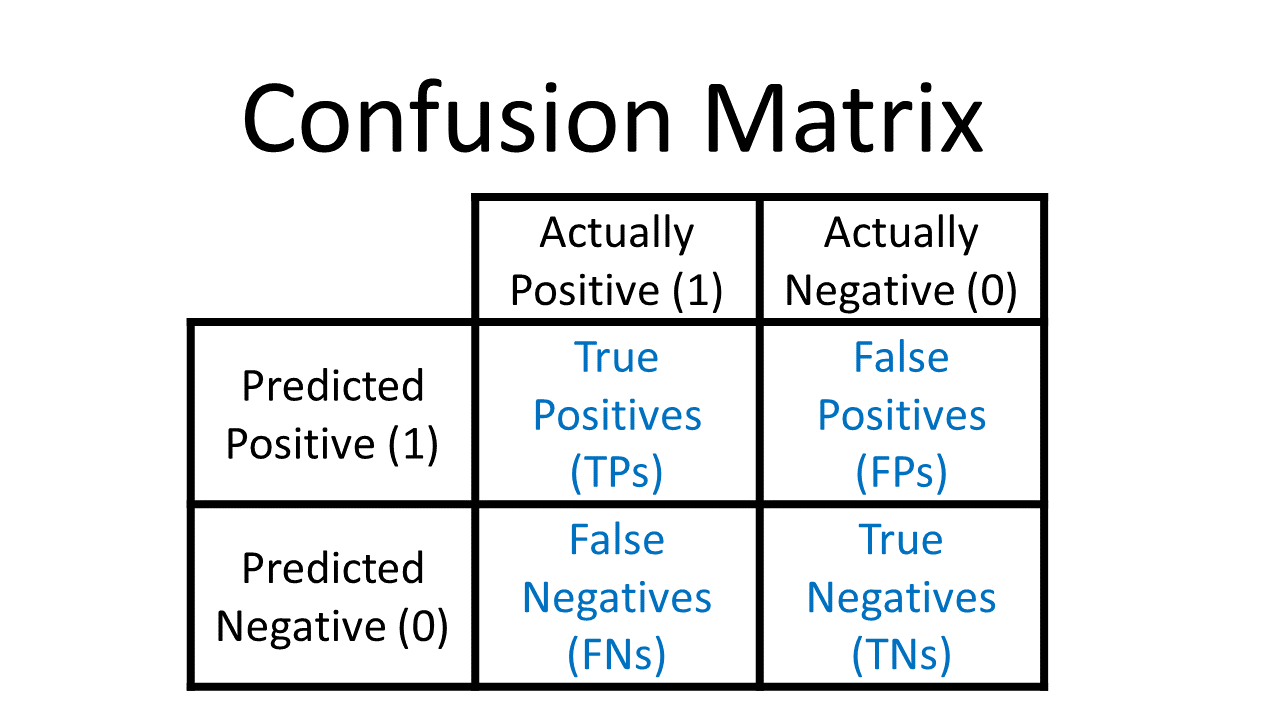
* Getting the coefficients of the model:

logmodel.coef\_

* Getting the probabilities of a datapoint belonging to each class:

logmodel.predict\_proba(X\_test)

**Classification metrics: Confusion Metrics and accuracy:**



* Accuracy: (True positives + True Negatives) / Total Predictions
* But accuracy values can sometimes be misleading especially for **imbalanced classes.** So, any classifier dealing with imbalanced classes has to confront the issue of the accuracy paradox. That is, the accuracy values present a rosier picture than the actual reality. For this reason we look at additional metrics too.
* **Recall:** When it is actually a positive case, how many of the predictions are correct.

**(True positive)/ Total Actual Positives**

* **Precision:** When the model predicts positive, how often is it correct.

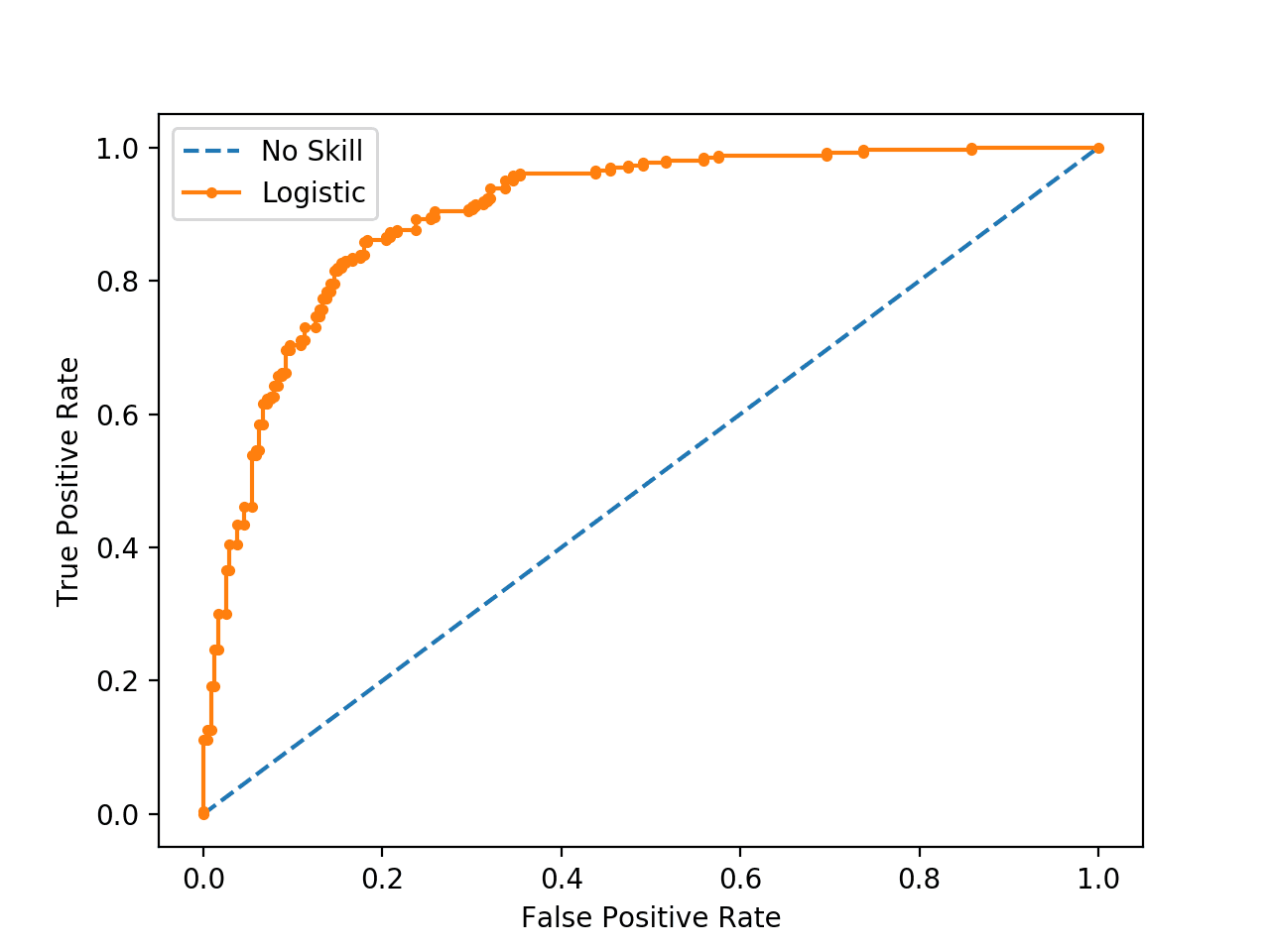
**(True positive)/Total Predicted Positives**

* **F1 score:** Harmonic mean of precision and recall.

**F = (2x precision x recall)/ (precision + recall)**

So, if either precision or recall is 0, F score goes to 0 indicating that this might be a case of accuracy paradox.

* **ROC Curve(receiver operator characteristic):** There can be a trade-off between true positives and false positives. We can adjust the threshold values so as to minimise the false negatives but resulting in more false positives.



* This is acceptable in certain situations where a false negative could prove fatal. (virus testing). False positive can always be rechecked.
* AUC curve: area under curve. Allows us to compare ROCs for different models.

**Checking the performance metrics of our logistic regression model**

from sklearn.metrics import accuracy\_score,confusion\_matrix,classification\_report

* **Accuracy**

accuracy\_score(y\_test,y\_pred)

* **Confusion matrix**

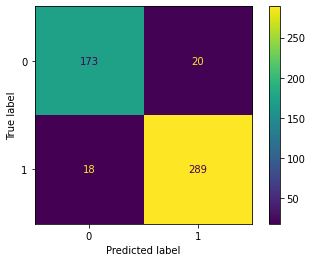
confusion\_matrix(y\_test,y\_pred)

This outputs an array of values. For easier interpretation we use the following:

from sklearn.metrics import plot\_confusion\_matrix

plot\_confusion\_matrix(model = logmodel,X\_test,y\_test)

For other classification models, we just need to change the model parameter.

****

**Classification report**

* This includes the recall, precision and f1 score values for each class. The mean an weighted mean can also be obtained through this metric to derive an idea whether the dataset is imbalanced.

print(classification\_report(y\_test,y\_pred))

**Precision and recall values**

from sklearn.metrics import precision\_score,recall\_score

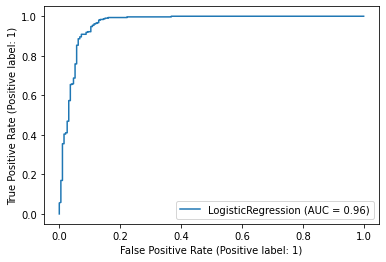
precision\_score(y\_test,y\_pred)

recall\_score(y\_test,y\_pred)

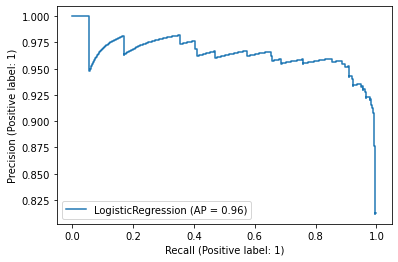
**Plotting the ROC and precision – recall curve**

from sklearn.metrics import plot\_precision\_recall\_curve,plot\_roc\_curve

plot\_roc\_curve(model=logmodel,X\_test,y\_test,ax=ax)



plot\_precision\_recall\_curve(logmodel,X\_test,y\_test)



**Multiclass logistic regression**

**K Nearest Neighbours**

* It assigns label to new data based on its distance to existing data points.
* For a given value of ‘k’, the algorithm picks up the ‘k’ nearest data points to the current data point
* **Handling a tie**: Let’s say k is even. For a new data point if the k/2 nearest points belong to class 1 and the other k/2 belong to class 2, there will be a tie leading to ambiguity. In scikit-learn, the answer will be the class that happens to appear first in the set of neighbours (the class of the closest point).
* **Choosing optimal k**: Two ways
* elbow method(this can be prone to manual discrepancy)
* cross validation
* Two main considerations:
* choosing optimal k
* Scaling features

**Code:**

**Using an arbitrary k value:**

from sklearn.neighbors import KNeighborsClassifier

KNN\_classifier = KNeighborsClassifier(n\_neighbors=1)

Training and predictions

KNN\_classifier.fit(scaled\_X\_train,y\_train)

y\_pred = KNN\_classifier.predict(scaled\_X\_test)

Performance metrics

from sklearn.metrics import confusion\_matrix,plot\_confusion\_matrix,classification\_report

confusion\_matrix(y\_test,y\_pred)

print(classification\_report(y\_test,y\_pred))

**Choosing and optimal K value:**

**Using elbow method (manual):**

from sklearn.metrics import accuracy\_score

test\_error\_rates=[]

for k in range(1,30):

    knn\_model = KNeighborsClassifier(n\_neighbors=k)

    knn\_model.fit(scaled\_X\_train,y\_train)

    y\_pred\_test = knn\_model.predict(scaled\_X\_test)

    test\_error = 1-accuracy\_score(y\_test,y\_pred\_test) #test error = 1- accuracy

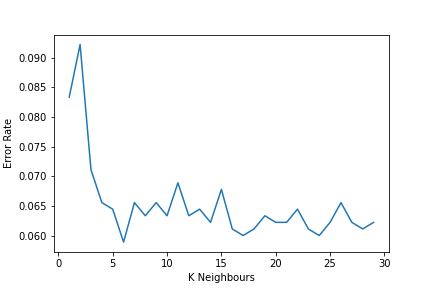
    test\_error\_rates.append(test\_error)

**Plotting this we obtain:**

plt.plot(range(1,30),test\_error\_rates)

plt.ylabel('Error Rate')

plt.xlabel('K Neighbours')

****

Observe the K-value where the curve stagnates or starts fluctuating rapidly (k=6) in this case. That will be the optimal value. This however can be tedious. Hence we go for the grid search cross validation method.

* A new method (not restricted to K nearest neighbours) is used here, called pipelining. We create a pipe object and specify the operations that need to be performed and then feed it to gridsearch.

**Creating pipe operator and specifying the operations:**

*Initialising the operations:(scaler and knn)*

scaler = StandardScaler()

knn = KNeighborsClassifier()

*Specifying the order of operations:*

operations =[('scaler',scaler),('knn',knn)]

*Creating the pipe object:*

from sklearn.pipeline import Pipeline

pipe =Pipeline(operations)

*Feeding this to the grid search model:*

from sklearn.model\_selection import GridSearchCV

k\_values = list(range(1,20))

param\_grid= {'knn\_\_n\_neighbors':k\_values,'knn\_\_algorithm':['auto','ball\_tree','kd\_tree','brute']}

* Care has to be taken while specifying the param\_grid. Normally the dictionary keys would be ‘n\_neighbors’ and ‘algorithm’.
* But when using pipe operators the convention is as follows:

**operation\_name(knn) + \_\_(two underscores) + parameter\_name(n\_neighbors/algorithm)**

full\_cv\_classifier = GridSearchCV(pipe,param\_grid,cv=5,scoring='accuracy')

full\_cv\_classifier.fit(X\_train,y\_train)

full\_cv\_classifier.best\_estimator\_.get\_params() #Obtaining the best parameters through cross validation

*Performance metrics*

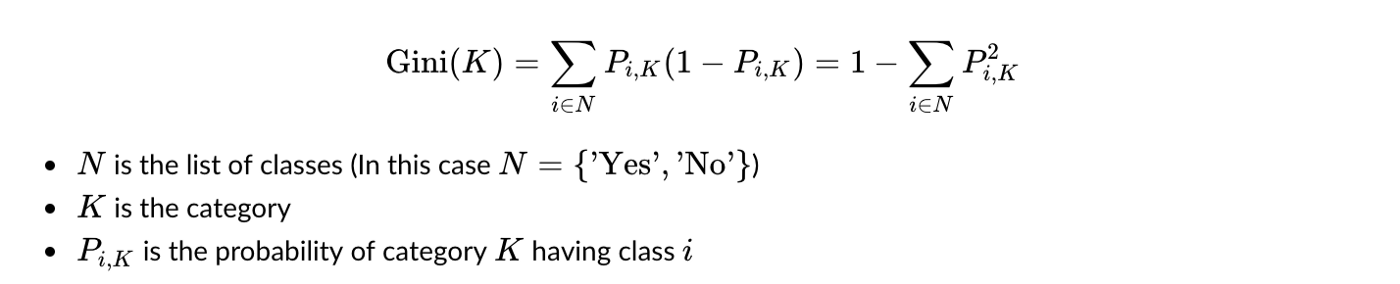
y\_pred = full\_cv\_classifier.predict(X\_test)

print(classification\_report(y\_test,y\_pred))

**Tree based methods**

**Decision Trees**

* **Gini impurity:** Measure of class uniformity.



* Maximum possible value is 0.5
* As the data is more pure(less impurity) the gini impurity values goes down. This means smaller gini impurity means that data is more of a single class.(Say there are classes A and B. case A has 250 items each of class A and B whereas case 2 has 350 of class A and 50 of class B) then:

**(gini impurity)case 2  < (gini impurity)case 1**

* If all data points belong to a single class then gini impurity will be 0.
* In tree based methods, we want to minimise gini impurity values on leaf nodes(which effectively implies we are separating out classes and achieving our goal of classification).

**Constructing decision tree based on gini impurity values:**

**For a binary categorical feature (calculating the gini impurity at each leaf node and taking the weighted average)**

* Consider we have two features X and Y .Let us take X to be the root node and have a split based on the outcomes of X.(Example let X be emails containing URL and the outcomes are YES or NO)
* Let Y be the mails being spam and the outcomes are YES or NO. In each leaf node resulting from the split performed above, note down the occurrences of feature Y outcomes.(3/4 mails on the left node might be spam while 1/7 mails on the right node might be spam)
* Now carry out the gini impurity calculations based on the formula above and obtain a value for the left and right nodes. Once this is done, get the weighted average of these gini impurities.

**Continuous numeric features (By considering varied threshold values)**

* Now let X be a continuous numeric data. The first step would be to sort the data.
* What follows will be an iterative process. We can consider value < some threshold value and calculate the gini impurity. This has to be tried for different threshold values based on the feature values and the value which yields the lowest gini impurity will be chosen as the root node.

**Multi-categorical features (by considering one category at a time or combination of categories)**

* Here X is a multi-categorical feature. Let X have values A,B and C. We consider these features one at a time and calculate gini impurity. We can also consider the combination of these features (two features considered at a time) and calculate the gini impurity and choose the combination which has the least impurity.
* In case of multiple features, the model selects the feature with the lowest gini impurity as the root node.
* Large trees could lead to overfitting. So we can add in a threshold gini decrease hyperparameter. The split will only be carried out if the threshold value is satisfied.

**Code**

Before going into the code part, here are few key points:

* Categorical data needs to be encoded
* Scaling is not needed since we aren’t comparing features and hence the units won’t matter

X = pd.get\_dummies(df.drop('species',axis=1),drop\_first=True)

y= df['species']

Either this or use one-hot-encoder to encode the categorical features. The drop\_first=True has been done to avoid the dummy\_variable trap which will already be taken care in sklearn one-hot-encoder.

*Train-test Split*

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test= train\_test\_split(X,y,test\_size=0.33,random\_state=101)

*Importing, training the model and obtaining predictions*

from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier()

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

*Performance metrics*

from sklearn.metrics import classification\_report,plot\_confusion\_matrix

print(classification\_report(y\_test,base\_preds))

plot\_confusion\_matrix(model,X\_test,y\_test)

*The following command helps us to obtain the relative importance of the features in determining the outcome:*

model.feature\_importances\_

*Which can be better visualised as a dataframe:*

pd.DataFrame(index=X.columns,data=model.feature\_importances\_,columns=['Feature\_Importance']).sort\_values('Feature\_Importance')

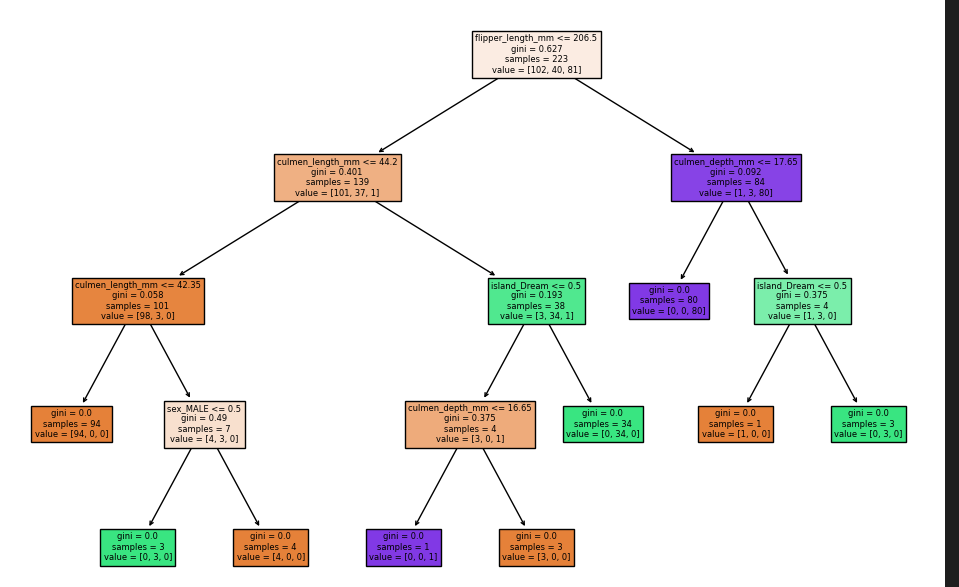
*Visualizing the decision tree:*

from sklearn.tree import plot\_tree

plt.figure(figsize=(12,8),dpi=100)

plot\_tree(model,feature\_names=X.columns,filled=True);

* If **feature\_names = X.columns** is not specified then the tree becomes difficult to interpret.
* **filled = True** colours the nodes based on feature used

**

**Tuning hyperparameters (‘maximum\_tree\_depth’** and **‘maximum\_leaf\_nodes’)**

* We can tune parameters like **‘maximum\_tree\_depth’** and **‘maximum\_leaf\_nodes’** to trade-off accuracy and tree complexity. Grid search has been used to accomplish the same:

max\_leaf = list(range(5))

max\_depth = list(range(5))

param\_grid = {'max\_depth':max\_depth,'max\_leaf\_nodes':max\_leaf}

from sklearn.model\_selection import GridSearchCV

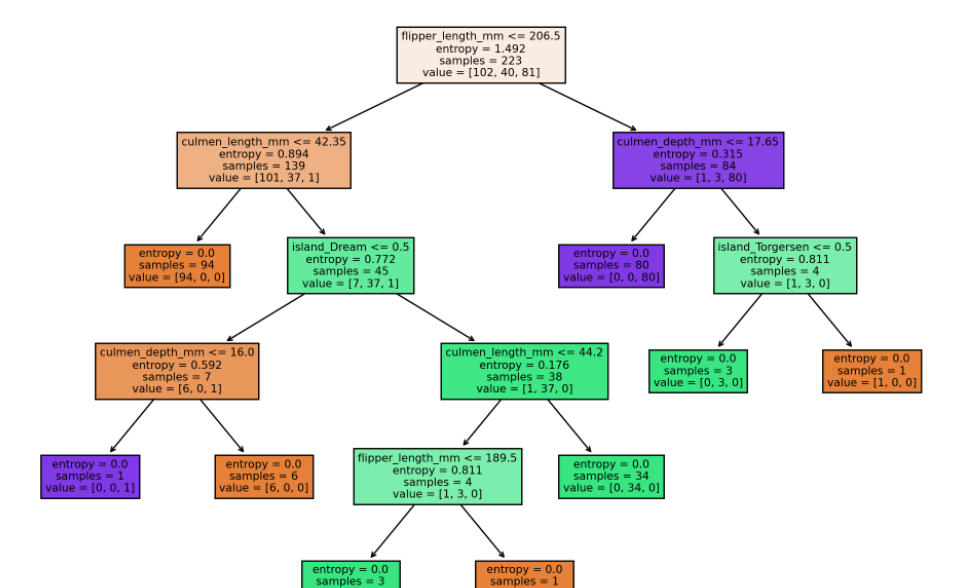
grid\_model = GridSearchCV(model,param\_grid=param\_grid,cv=5,scoring='accuracy')

grid\_model.fit(X\_train,y\_train)

* Maximum tree depth limits can be used to avoid overfitting, because a tree with large number of levels might sometimes imply overfitting whereas maximum leaf nodes can help in obtaining a simpler tree which would be easier to interpret.
* While these splits have been carried out based on the value of gini impurity, entropy values can be used for splitting too:

entropy\_tree = DecisionTreeClassifier(criterion='entropy')

entropy\_tree.fit(X\_train,y\_train)

****

**Ensemble Methods**

* Based on the principle that by aggregating the predictions of a diverse group of predictors we will often get better predictions than with the best individual predictor.
* Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is by training them using very differernt algorithms. This increases the chance that they will make different types of errors improving the ensemble’s accuracy.

**Random Forest Methods**

* Drawbacks/ points to be remembered when building decision trees:
* No guarantee of using all features (sometimes the features that aren’t used might hold some useful information)
* Root node will always be the same.
* Root node has a huge influence over tree
* The basic idea here is to **create subsets of randomly picked features at each potential split.**
* When the output is categorical, we tally up the votes/outcomes of the trees within the random forests. When the output is continuous, we average the individual outputs. In short,

**Classification: Most voted output class**

**Regression: Average predicted output value**

**Key hyperparameters**

* **Number of estimators** : **(Amount of decision trees to use)**
* It is observed that even for a very large number of decision trees, **the decision trees don’t overfit.**
* As the number of trees increases, the resultant trees are likely to become highly correlated and we would simply be duplicating trees that have already been created.
* So this is one of the key highlights of decision trees. *We can be lenient with the number of trees without having to worry about overfitting.*
* It is suggested to start with 100 as default and then grid search for higher values
* **Features :** (**Features to include in each subset)**
* No hard and fast rule about the value to be chosen. Start with sqrt(N) and grid search for other possible values(N/3)
* **Bootstrap samples(randomly select rows of data along with features)**
* Process of random sampling with replacement.
* In addition to choosing a random subset of features, we randomly select rows of data too. Since this is bootstrapping, there might be some duplicates.
* This is meant to reduce correlation between decision trees. Since we are training trees on different features and rows of data it helps us better differentiate the trees.
* **Out –of-bag error(Use data not used by trees while training to obtain their performance metrics)**
* The ‘bag’ here stands for bootstrapping and aggregated prediction.
* **Out of bag samples :** Due to bootstrapping of data rows, some samples will not be used for constructing trees and we could use that data to get performance test metrics for trees that did not use that data
* The performance metric thus obtained is called the **out of bag score**. It is just another way of measuring performance but doesn’t really affect model performance.

**Code**

from sklearn.ensemble import RandomForestClassifier

rfc = RandomForestClassifier(n\_estimators=10,max\_features='auto',random\_state=101)

The max features parameters can take in a variety of inputs including integer, float and string values. Each value has its own interpretation.

rfc.fit(X\_train,y\_train)

y\_pred = rfc.predict(X\_test)

**Tuning the hyperparameters to obtain best performance**

from sklearn.model\_selection import GridSearchCV

from sklearn.ensemble import RandomForestClassifier

rfc = RandomForestClassifier()

*Hyperparameters to be tuned\*

n\_estimators = [64,100,128,200]

max\_features = [2,3,4]

bootstrap = [True,False]

oob\_score =[True,False]

* This might throw up some warnings, especially in cases when bootstrap= true and oob\_score =false.

An alternative way of determining the optimal values can be plotting out errors vs number of estimators used as well as misclassifications vs number of estimators.

errors =[]

misclassifications = []

for n in range(1,200):

    rfc = RandomForestClassifier(n\_estimators=n,max\_features=2)

    rfc.fit(X\_train,y\_train)

    pred = rfc.predict(X\_test)

    error = 1- accuracy\_score(y\_test,pred)

   #in the following line we are comparing elements of y\_test and pred

   #and the ones that aren't equal are considered as misclassifications

    n\_missed = np.sum(pred != y\_test)

    errors.append(error)

    misclassifications.append(n\_missed)

**Regression using random forests**

* Sometimes like in the case of linear regression, the performance metrics may paint a rosier picture than the actual ground reality. So it is better to visualize the prediction output to get a clearer idea.
* The following function is useful for comparing the model performance for various regression models. It trains, predicts, calculates the performance metrics and also plots the result.

def run\_model(model,X\_train,y\_train,X\_test,y\_test):

    #Fit model training

    model.fit(X\_train,y\_train)

    #Get metrics

    preds = model.predict(X\_test)

    rmse = np.sqrt(mean\_squared\_error(y\_test,preds))

    mae = mean\_absolute\_error(y\_test,preds)

    print(f'MAE: {mae}')

    print(f'RMSE: {rmse}')

    #Plot results

    signal\_range = np.arange(0,100)

    output = model.predict(signal\_range.reshape(-1,1))

    plt.figure(figsize=(12,8),dpi=200)

    sns.scatterplot(x='signal',y='density',data=df)

    plt.plot(signal\_range,output)

*Checking the model performance for polynomial regression*

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import PolynomialFeatures

pipe = make\_pipeline(PolynomialFeatures(degree=6),LinearRegression())

run\_model(pipe,X\_train,y\_train,X\_test,y\_test)

Difference between **pipeline** and **make\_pipeline**

* Earlier during KNN classifier, Pipeline() was used whereas here we are importing make\_pipeline()
* In Pipeline() we specify the name for each operator explicitly whereas in make\_pipeline it is generated automatically

Example: Pipeline() : operations =[('scaler',scaler),('knn',knn)]

make\_pipeline: operations =[scaler,knn]

We would like to know the names of the steps in cases where we intend to tune model parameters for example when we are specifying the param\_grid in GridsearchCV.

* For Support vector machine regression, we will need to use gridsearch to obtain the values for ‘C’ and gamma.
* Plotting results for different values of ‘k’ in K-Nearest neighbors regression:

from sklearn.neighbors import KNeighborsRegressor

k\_values=[1,5,10,20,30]

for k in k\_values:

    model = KNeighborsRegressor(n\_neighbors=k)

    run\_model(model,X\_train,y\_train,X\_test,y\_test)

*Random Forest Regression*

from sklearn.ensemble import RandomForestRegressor

rfr = RandomForestRegressor(n\_estimators=10)

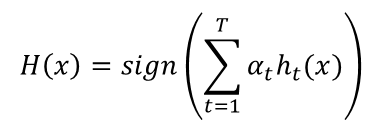
* If needed, we can cross validate for ideal values of n\_estimators, max\_features , bootstrap and other hyperparameters.

**Boosting Methods**

* Basic idea is that a combination of estimators with an applied coefficient could act as an effective ensemble estimator. So in simple words, **we combine weak learners(simple models) to create a stronger model.**
* This is not restricted to decision trees. This can effectively be extended to any machine learning algorithm.

**Adaboost**

* Each subsequent tree/model learns from the tree/model that came before them.



T – Number of models we are aggregating

h(x) – hypothesis of a particular tree

alpha – weightage assigned to that hypothesis. Depends on how accurate the hypothesis is

**Algorithm:**

* Choose a hypothesis of weak learner (h(x)) which minimises the error term for a particular tree. The process of minimising the error involves assigning higher weightages to data points misclassified by the decision tree and low weightages to ones which were classified correctly.
* Based on this minimized error value, obtain the alpha(weight) for that particular hypothesis
* Once this is done, we add it to the existing ensemble.
* Next update the weights which will be used by the next weak learner. So the next weak model won’t be learning based on the initial weights but instead will be depending on the weights assigned during the previous weak learner cycle. We repeat this step T times (T – number of models in ensemble)

***In summary:***

* Adaboost uses an ensemble of weak learners that learn slowly in series.
* Certain weak learners(models) have a greater say in the final output based on the relative weightage assigned to them(the alpha term)
* Each subsequent tth weak learner is build using a reweighted data set from the t-1 weak learners that came before it.

While decision trees and random forests help us determine the relative importance of features towards the final outcome, adaboost specifically has stumps focusing on one feature at a time and thus facilitating better understanding.

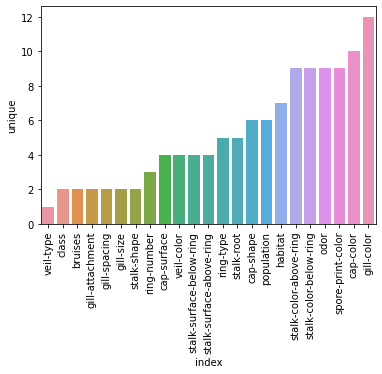
**Code and Intuition**

*Plotting the number of unique values per feature*

feat\_uni = df.describe().transpose().reset\_index().sort\_values('unique')

sns.barplot(data=feat\_uni,x='index',y='unique')

plt.xticks(rotation =90);

**

#Encoding the categorical features

#drop first=True in order to avoid the dummy variable trap

X = pd.get\_dummies(X,drop\_first=True)

*Model Building*

from sklearn.ensemble import AdaBoostClassifier

model = AdaBoostClassifier(n\_estimators=1)

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

* **base\_estimator** parameter is used to choose the model from which the boosted ensemble will be built
* By default the estimator is a decision tree with a depth 1

Similar to random forests and decision trees, we can obtain the relative importance assigned to individual features:

model.feature\_importances\_

Which can be better interpreted in a data-frame:

features = pd.DataFrame(index=X.columns,data=model.feature\_importances\_,columns=['Importance'])

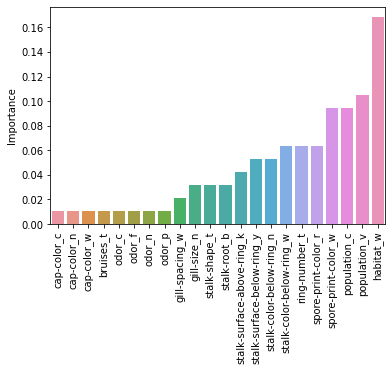
Filtering out features with non-zero importance values:

importances = features[features['Importance']>0]

Plot:

sns.barplot(data=importances.sort\_values('Importance'),x=importances.index,y='Importance')

plt.xticks(rotation=90);



**Determining the best value for number of estimators**

error\_rates = []

for n in range(1,len(X.columns)+1):

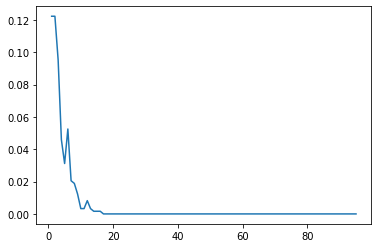
    model = AdaBoostClassifier(n\_estimators = n)

    model.fit(X\_train,y\_train)

    preds = model.predict(X\_test)

    err = 1 - accuracy\_score(y\_test,preds)

    error\_rates.append(err)

****

**The value where the error saturates is the optimal value. The same can be obtained through grid search too.**

**Gradient Boosting Theory**

* Larger trees are allowed in gradient boosting(as opposed to stumps in adaboost)
* Learning rate coefficient is same for each new model in series and not unique to each subsequent model like in Adaboost.
* It is robust to overfitting.
* We optimize the series of trees by learning on the residuals forcing the subsequent trees to attempt to correct for the error in the previous trees.
* The trade-off is learning rate. The choice of learning rate (between 0-1) has a great say in the computational time as well as the relative importance of each subsequent tree in the final outcome.

**Algorithm**

* Create an initial model: f0. Use this model to obtain initial predictions.
* Obtain the error for these predictions.

e = y- f0

* Train another model on this obtained error.
* Create a new prediction:

F1 = f0 + (learning\_rate)\*f1

* Repeat as needed:

Fm = fm-1 + (learning\_rate)\*fm

**Code and Intuition**

There are a number of hyperparameters to be considered and hence we go for grid search.

from sklearn.ensemble import GradientBoostingClassifier

from sklearn.model\_selection import GridSearchCV

param\_grid = {'n\_estimators':[50,100],'learning\_rate':[0.1,0.05,0.2],'max\_depth':[3,4,5]}

gb\_model = GradientBoostingClassifier()

grid = GridSearchCV(estimator=gb\_model,param\_grid=param\_grid)

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

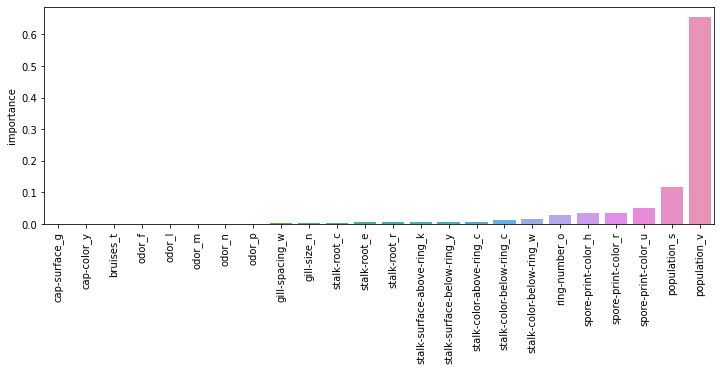
* Key thing, while obtaining the feature importance through a grid search model, first call best estimator and then the feature importances **(grid.best\_estimator\_.feature\_importances\_)**
* This can again be better visualised through a dataframe

imp\_feat = feat\_importances[feat\_importances['importance']>0.0005]

plt.figure(figsize=(12,4))

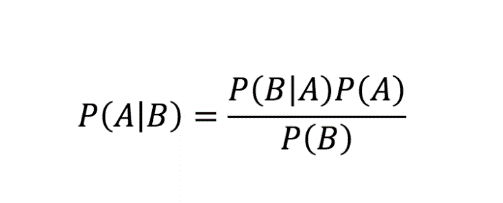
sns.barplot(data=imp\_feat.sort\_values('importance'),x=imp\_feat.index,y='importance')

plt.xticks(rotation=90);

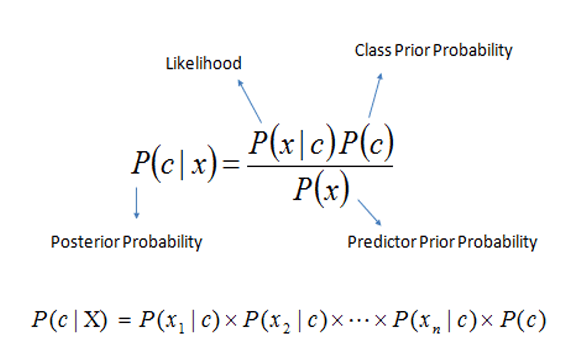


**Naive Bayes and Natural Language Processing**

**Naive Bayes Approach  
Bayes Theorem**



* We can model the probability of belonging to a class(Ck) given a vector of features(X)



* In order for this chain rule to apply, we need to assume that all the x features are mutually independent of each other. Although this might not necessarily be true, it is important to proceed further with the computations.

**Unsupervised Learning**

* Two chief approaches:
* Clustering: Cluster of group together similar data points to discover possible labels for clusters
* Dimensionality reduction : Reduce the number of features by combining them into new components
* It is much harder to compare unsupervised algorithms against each other due to lack of ground truth based performance metrics(like RMSE or accuracy)

**K Means Clustering**

* At times, it might be confusing to arrive at the ideal number of clusters to be chosen.

Important points:

* We use features to decide which points are similar to other points.
* There is no correct y label with which we can compare our cluster results.

**Theory**

* Properties each point must satisfy:
* Each point must belong to a cluster.
* Each point must only belong to a single cluster.
* Step 1: Choose the number of clusters to create(K values)
* Step 2: Randomly select K distinct data points.
* Step 3: Assign each remaining point to the nearest ‘cluster’ point.
* Step 4: Calculate the centre of the cluster points(mean value of point vectors)
* Step 5: Now assign each data point to the nearest cluster centre.

**Random Initiailisation Trap**

* The initiliazing of centroids at the beginning could sometimes influence the final outcome and might lead to results which might not necessarily be correct. The solution to this is to use K Means++.

**Code**

We don’t need to worry about dummy variable traps because there are no coefficients attached to the features in unsupervised learning.

X = pd.get\_dummies(df)

As we have no idea about the output labels, test-train splits don’t make sense either. We however scale our data since the distance metric is involved.

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

scaled\_X = scaler.fit\_transform(X)

Importing the model and choosing number of clusters

from sklearn.cluster import KMeans

model = KMeans(n\_clusters=2)

Analogous to the **‘fit\_transform’**  method calls used in supervised learning algorithms, we go for **‘fit\_predict’**  in unsupervised learning. This method will output an array of cluster labels.

cluster\_labels = model.fit\_predict(scaled\_X)

Not much importance must be assigned to the exact values of these cluster labels since these are arbitrary values being chosen by the model. Now we add back this clustering result to our original dataset.

X['Cluster'] = cluster\_labels

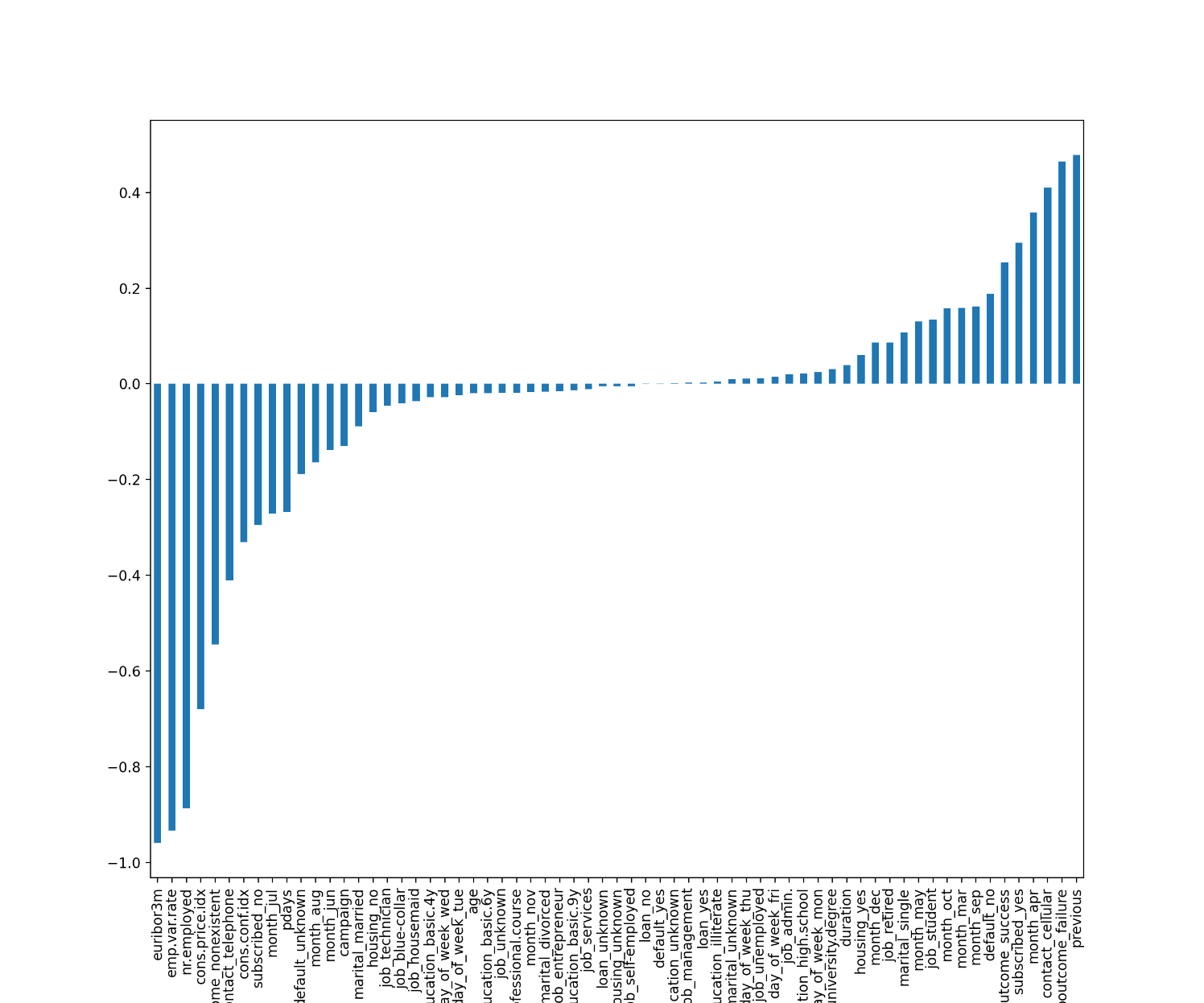
**Interpreting the results**

* The next important part would be to interpret the clustering results. And a good way of doing that would be by checking how much each feature is correlated to the cluster column.
* Once we obtain this info, we can approach a domain expert and try to know what these clusters may stand for

plt.figure(figsize=(12,10),dpi=200)

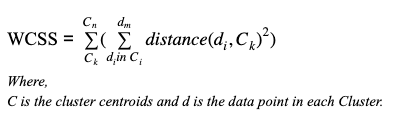
X.corr()['Cluster'].iloc[:-1].sort\_values().plot(kind='bar')

plt.xticks(rotation=90);



**Choosing the ideal K value**

* Since we can't go for normal error/performance metrics, we can measure the sum of distances from points to cluster centres for a given K clusters.
* To punish the points farther away from the cluster centres , we go for the squared sum of distances**(sum of squared distances(SSD))**



* Next consider K+1 clusters and calculate the SSD. Ultimately for a sufficiently large value of K this will go to zero. **Our primary goal would be determine the rate of decline of SSD for every subsequent value of K and not just the absolute value.**
* We keep track of this SSD value for a range of different K values and check where the rate of reduction in SSD begins to decline. This basically implies that the addition of a new cluster at that point isn't justifying the performance improvement.

Function to obtain ssd values for incremental values of K:

ssd =[]

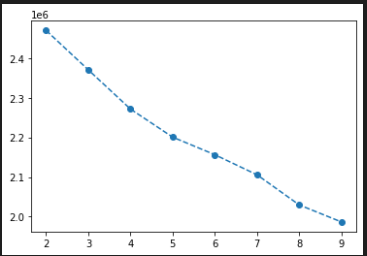
for k in range(2,10):

    model = KMeans(n\_clusters=k)

    model.fit(scaled\_X)

    ssd.append(model.inertia\_)

**model.inertia\_** - distance between data point and the cluster centre. This is the call that will yield us the value of SSD for each value of K



Another way of checking the differences in SSD between subsequent levels:

pd.Series(ssd)

pd.Series(ssd).diff()

**Colour Quantization**

* Quantization is a process that reduces the number of distinct colours used in an image, usually with the intention of preserving the originality of the image. Basically it is a lossy compression algorithm.
* Grey scale images are stored in a way that each pixel has either 1(for white) or 0(for black).
* For colour pictures however, a 3D array approach is used to store them.

H x W x C

H- height of the image

W – width of the image

C- a 3D array again which has the colour channel values(RGB) for each pixel.

* So our goal is to average out the colour streams present in the picture while also making sure the originality is preserved as much as possible.
* In order to operate upon this image, we can reshape the image from a 3D array to a 2D array with feature set with features R,G,B:

Before : H with R,G,B channels and W with R,G,B channels

After : HxW with R,G,B channels

* Then we choose a K value of colours and use K means clustering to create labels. Basically this is us specifying how many distinct colours we need in our resulting image.
* The clusters thus obtained must have a cluster centre and these centres are in terms of original feature space(R,G,B). This means each cluster centre is an average(R,G,B) value that can be used for reassignment.
* The original pixel values are replaced by these values and the quantized image is obtained

**Code**

Since we are dealing with images (.jpg or .png) files, we use the **matplotlib.image** library to read the images and save them as numpy arrays.

import matplotlib.image as mpimg

image\_as\_array = mpimg.imread('F:\\hard\_disk\\New folder\\Machine\_Learning\\K\_means\_clustering\\palm\_trees.jpg')

As mentioned previously, this will be a 3D array which needs to be converted to a 2D one in order to proceed with our K means clustering operation:

(h,w,c) =  image\_as\_array.shape

image\_as\_array2d = image\_as\_array.reshape(h\*w,c)

Training the model and obtaining the cluster labels:

from sklearn.cluster import KMeans

model = KMeans(n\_clusters=8)

labels = model.fit\_predict(image\_as\_array2d)

Now, the next step would be to obtain the cluster centre for each cluster.

rgb\_codes = model.cluster\_centers\_.round(0).astype(int)

* These values will be in decimal form which are rounded off to integers since we need it in RGB format. Basically in our case, we have 8 clusters which means there are 8 RGB combinations and all the pixels will be assigned to one of these clusters depending on their proximity.
* We proceed to replace each pixel value by the cluster centre to which it is mapped and reshape it to get back the 3D array representation.

quantized\_image = np.reshape(rgb\_codes[labels],(h,w,c))

**Hierarchial Clustering**

* Key feature is that it is not necessary to choose cluster amount before running the algorithm like in K means clustering.
* There are two approaches through which the points are divided into potential clusters:
* **Agglomerative approach:** Each point begins as its own cluster , then the clusters are joined
* **Divisive approach :** All points begin in the same cluster, then clusters are split
* Following is the approach that we adapt:
* Compare data points to find ones similar to one another. Merge these into a cluster.
* Then check for similar clusters and merge again.
* Repeat the process until all the points are in a single cluster.
* **Similarity metric**: Measuring distance between two data points. Default choice is euclidean distance.
* Using MinMaxScaler would mean that all the observations will be between 0 and 1. This would mean that the maximum possible distance between features would be 1.
* **Dendogram:** It is a plot displaying all the potential clusters. This is the key visual representation that lets us decide the number of clusters to be chosen.
* The lengths of then links connecting various clusters/observations along the y-axis play a key role in determining the extent to which they are similar. Longer the lengths, higher the distance between the observations.
* We can also obtain a varying number of clusters based on where we **‘slice´** the dendogram.
* **Linkage:** This is the criterion that is set while merging clusters. The algorithm works towards minimizing this criterion while checking for similarities between clusters.

Some key parameters:

* **ward :** this linkage parameter focuses on minimizing the variance between the observations between merged
* **average :** Uses average distances between two sets
* **Maximum** and **Minimum** between all observations of the two sets.

**Code and Intuition**

Since we are going for Euclidean distance metric, we opt for min max scaling so to obtain a fair estimate of the maximum distance between the datapoints

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

scaled\_data = scaler.fit\_transform(df\_w\_dummies)

Converting this back to a dataframe:

scaled\_df = pd.DataFrame(scaled\_data,columns= df\_w\_dummies.columns)

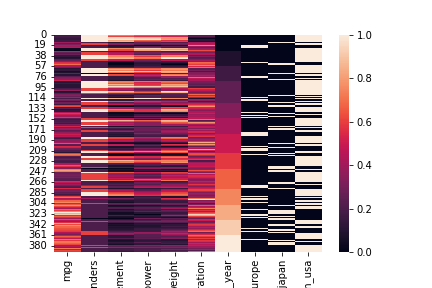
**Approaches to obtain the ideal number of clusters**

1. **Using seaborn Clustermaps**

Visuazlizing the min-max scaled values:

sns.heatmap(scaled\_df)

plt.savefig('scaled\_values.png')

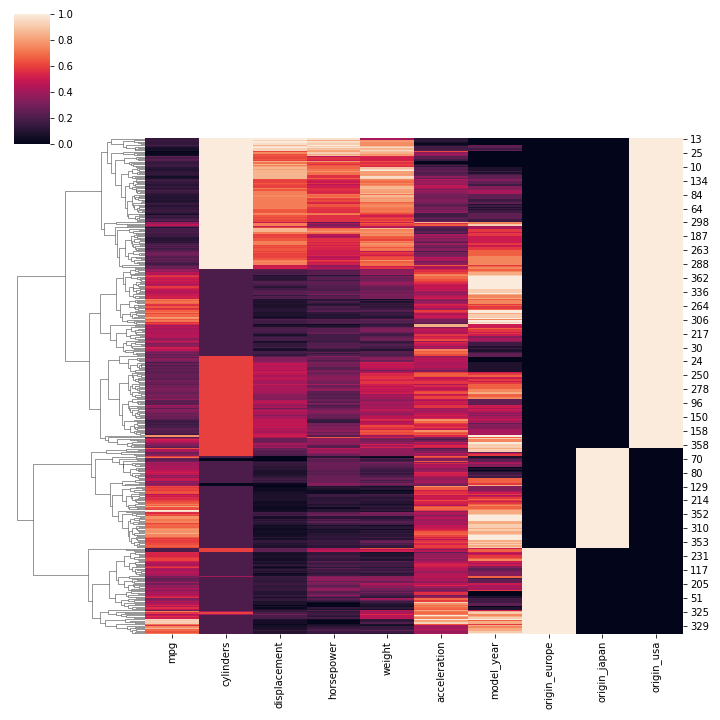


**Dendrogram Visualization**

The important thing to be noted is that we are interested in clustering the rows(data points) and not the features.

sns.clustermap(scaled\_df,col\_cluster=False)

In order to only visualise the dendrograms for features(columns) ‘**row\_cluster=False’.**



While this provides a fair estimate of how the clustering is being carried out, this approach is only suitable for smaller datasets. Thus we go for the next approach.

1. **Using sklearn AgglomerativeClustering()**

Model building and obtaining cluster labels:

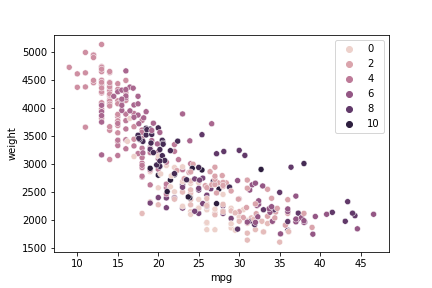
sns.clustermap(scaled\_df,col\_cluster=False)

model = AgglomerativeClustering(n\_clusters=4)

cluster\_labels = model.fit\_predict(scaled\_df)

We can try to have a scatterplot between features and colour them based on the cluster labels assigned to them to check if there is any observable pattern:

sns.scatterplot(data=df,x='mpg',y='weight',hue=cluster\_labels)



In the previous model call, we explicitly specified the number of clusters. But one of the advantages of hierarchial clustering is that this may not necessarily be the case. This is where the ‘**distance\_threshold’** parameter comes in.

model = AgglomerativeClustering(n\_clusters=None,distance\_threshold=0)

* **‘distance\_threshold’** is the distance above which clusters won’t be merged.
* So by setting **distance\_threshold** as zero, we are saying that even if the distance between teh data points are 0 then dont merge them into a cluster

cluster\_labels = model.fit\_predict(scaled\_df)

In order to better visualise the dendrograms, we go for scipy libraries:

from scipy.cluster.hierarchy import dendrogram

from scipy.cluster import hierarchy

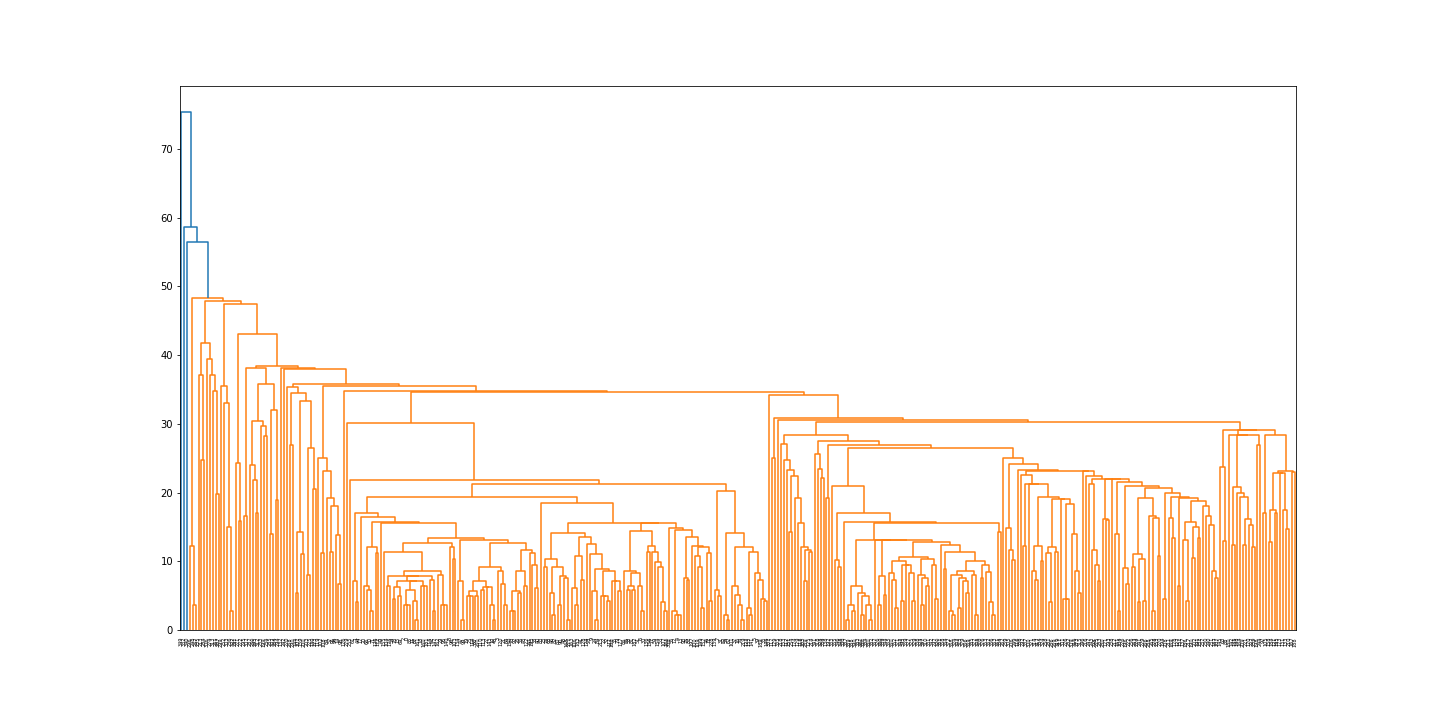
linkage\_matrix = hierarchy.linkage(model.children\_)

* Linkage matrix results are in the form [cluster/observation1, cluster/observation2, distance\_between\_them,number\_of\_points\_in\_cluster]

**[67. , 161. , 1.41421356 , 2. ]**

* It can be noticed that as we go down in this matrix, the distance between the points and the number of points in cluster keeps increasing progressively

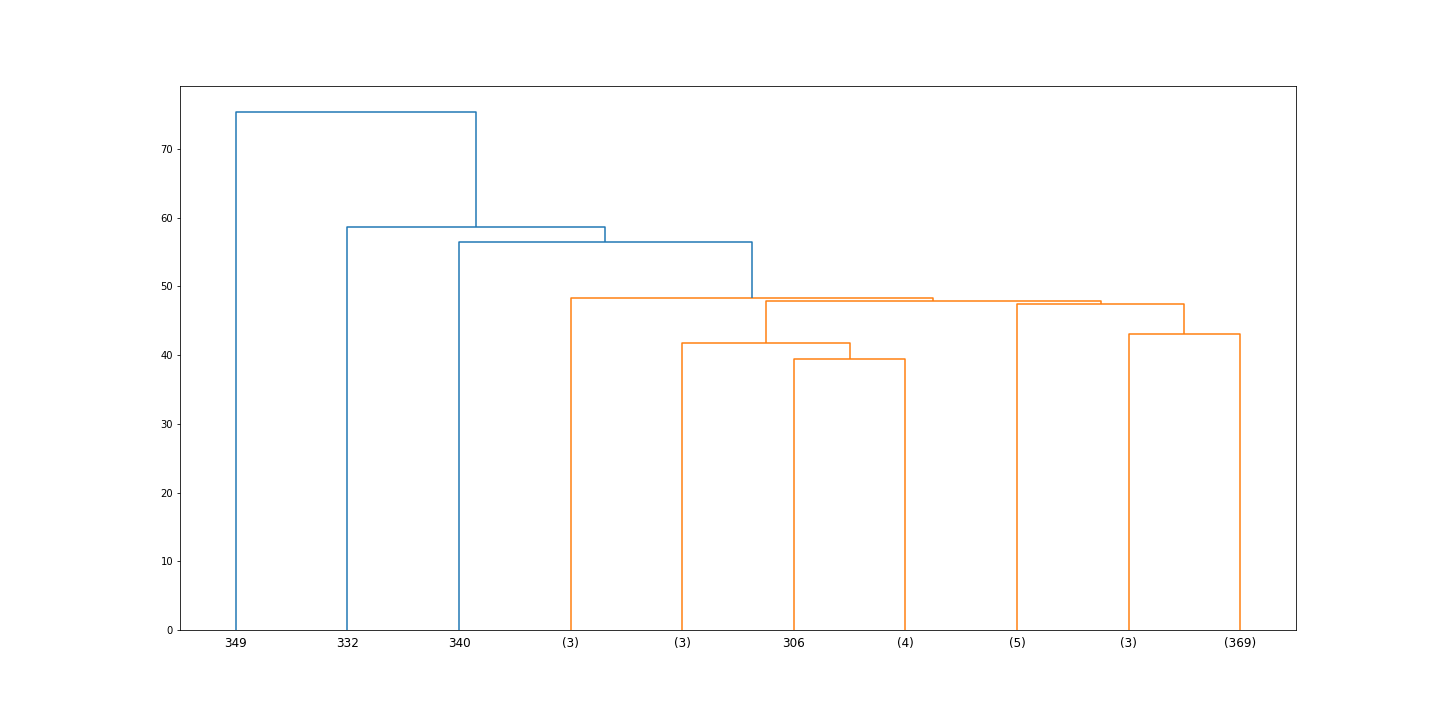
The dendrogram function takes in the linkage matrix as its input and produces the visualisation.



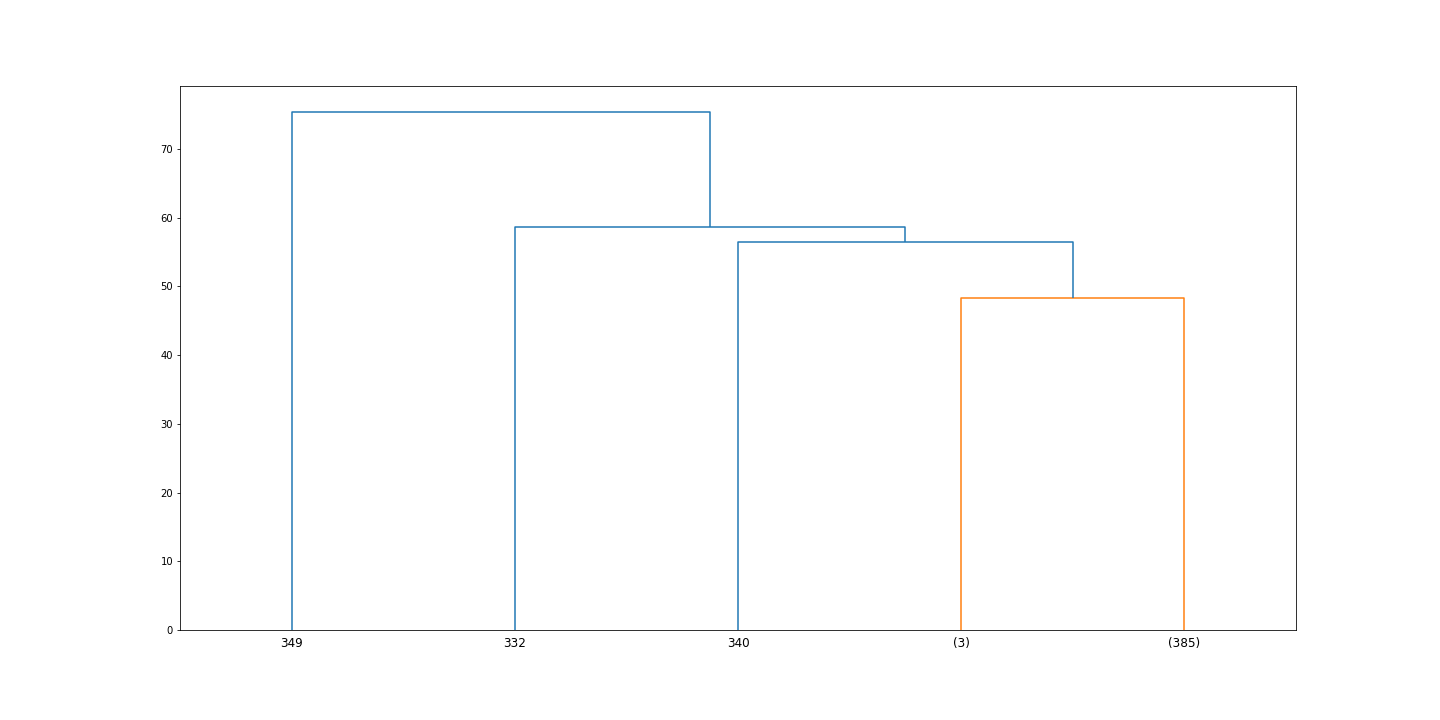
While this is a super informative depiction, interpretation can be an issue. It is for this reason we use the **'truncate\_mode’** parameter to condense the dendogram. We can specify two main values to this hyperparameter:

* lastp- The last p non-singleton clusters formed in the linkage are the only non-leaf nodes in the linkage
* level - No more than p levels of the dendrogram tree are displayed

**lastp**



**level**

****

* By exploring various values for the truncation hyperparameters we can try to arrive at the number of clusters to be used.However, this visual approach might not be useful for large datasets. Hence we lay emphasis on the euclidean distances.
* We go for tuning the **distance\_threshold** parameter instead of providing the number of clusters. To do this we need a range of values over which the **distance\_threshold** can be varied.
* Thereotical Max distance possible = sqrt(N\_features)

Explanation: Since we are using min max scaling, the worst case scenario would be (1,1) and (0,0) in which case the distance would be sqrt(2). So for N features it would be sqrt(N)

np.sqrt(len(scaled\_df.columns))

Now, obtaining the max distance between the actual datapoints in our dataset

scaled\_df['mpg'].idxmax()

scaled\_df['mpg'].idxmin()

car\_a = scaled\_df.iloc[320]

car\_b = scaled\_df.iloc[28]

distance = np.linalg.norm(car\_a-car\_b)

From the values obtained, it is fair to assume that 2 or 3 would be a reasonable estimate for the distance threshold parameter

model = AgglomerativeClustering(n\_clusters=None,distance\_threshold=2)

cluster\_labels = model.fit\_predict(scaled\_df)

Obtaining the number of clusters into which the data has been split into:

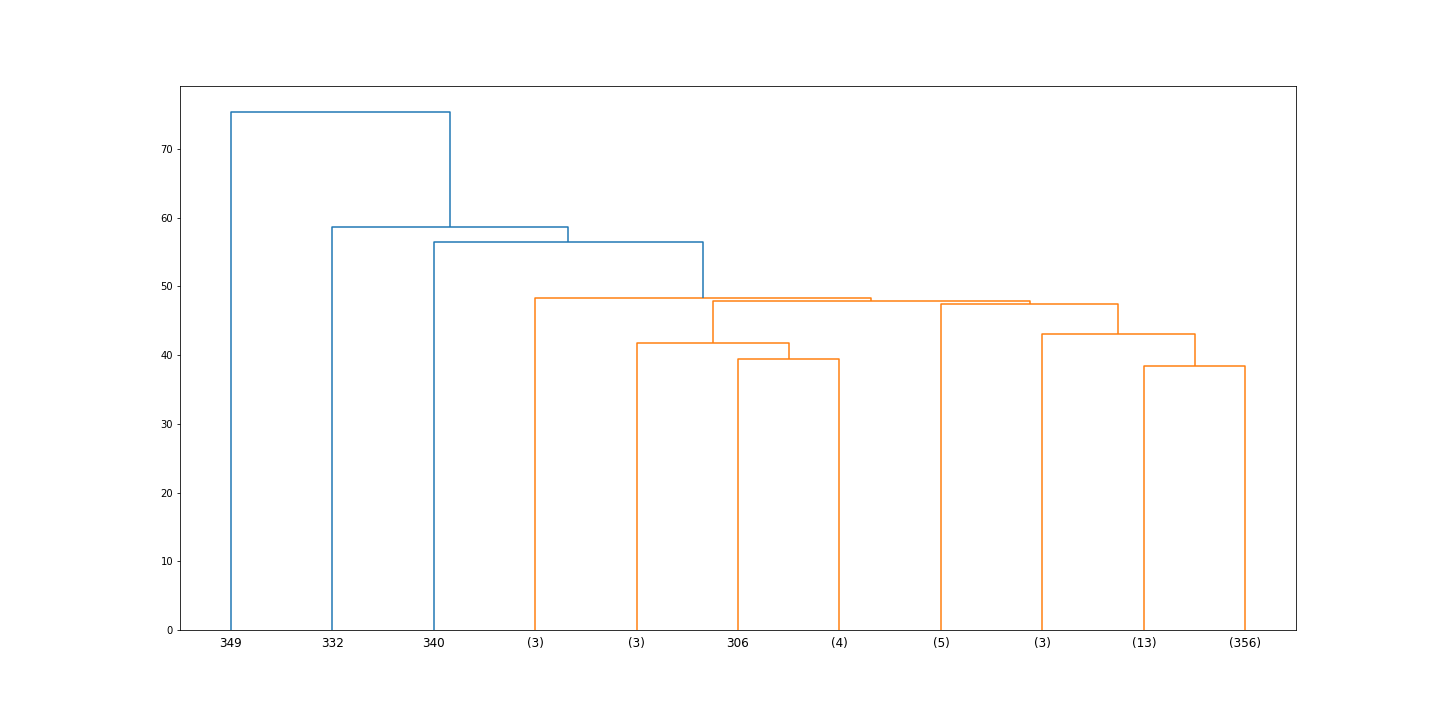
len(set(cluster\_labels))

linkage\_matrix = hierarchy.linkage(model.children\_)

p value is the number of clusters that we obtained just above:

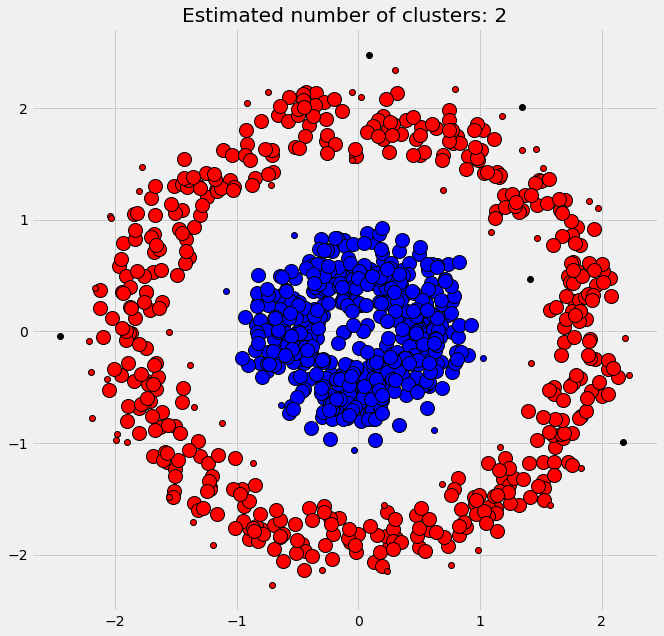
plt.figure(figsize =(20,10))

dendro = dendrogram(linkage\_matrix,truncate\_mode='lastp',p=11)



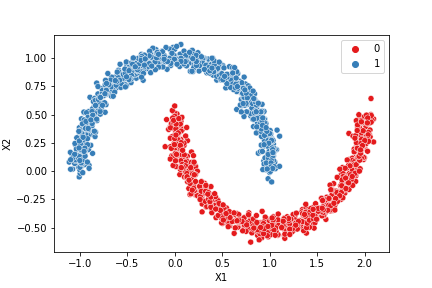
**DBSCAN(Denisty Based Spatial Clustering of applications with noise)**

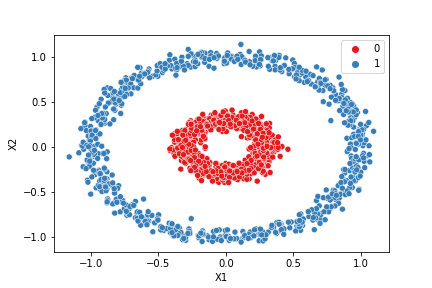
* Focuses on using density of points as its main factor for assigning cluster labels.
* We use 2 main hyperparameters:
* Epsilon: Distance extended from a point
* Minimum number of points: Minimum number of points in an epsilon distance
* DBSCAN point types:
* Core points: Point with minimum number of points in epsilon range. In most versions, the core point is included while counting the minimum points.
* Border point : In epsilon range of core point, but does not contain minimum number of points in epsilon-neighbourhood
* Outlier: Cannot be reached by points in a cluster assignment.
* Process for assigning clusters:
* Pick a random point that has not yet been assigned.
* Determine the point type.
* Once a core point has been found, add all directly reachable points to the same cluster as core.
* Repeat untill all points have been assigned to a cluster or as an outlier.



* Epsilon Intuition: Increasing epsilon allows more core points, which means more border points and less outlier points. This would effectively mean we’d have lesser number of clusters.
* Decreasing epsilon on the other hand would create more unique clusters.
* We can arrive at the ideal value by elbow method again. Possible plots can be epislon value vs number of outliers or even percentage of points labelled as outliers.
* This however will be largely dependent on the dataset being used.
* Minimum number of points intuition: Increasing this threshold would cause more points to be considered as unique outliers.
* Again, this value can be obtained through elbow point.
* It is useful to create potential new small clusters instead of complete outliers.

**Cases where K means clustering fails:**

****

****

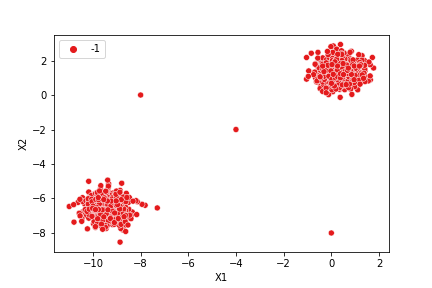
from sklearn.cluster import DBSCAN

model = DBSCAN()

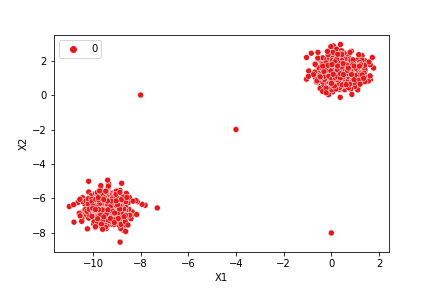
labels = model.fit\_predict(moons)

**Tuning epislon and minimum number of points parameters using elbow method:**

Consider a very small value of epsilon. It is noticed that almost all the points are classified as outliers since the threshold distance is so less



For a large value of epsilon however, all the points are merged and classified as a single cluster



Finding number of points classified as outliers

np.sum(dbscan.labels\_ == -1)

Percentage of total points labelled as outliers

100 \* np.sum(dbscan.labels\_ == -1) / len(dbscan.labels\_)

Tuning epsilon:

outlier\_percent = []

number\_of\_outliers =[]

for eps in np.linspace(0.001,7,200):

    dbscan = DBSCAN(eps=eps)

    dbscan.fit\_predict(two\_blobs\_outliers)

    #Total outliers

    number\_of\_outliers.append(np.sum(dbscan.labels\_ == -1))

    #Percentage of points classified as outliers

    perc\_outliers = 100 \* np.sum(dbscan.labels\_ == -1) / len(dbscan.labels\_)

    outlier\_percent.append(perc\_outliers)

Tuning minimum number of points:

outlier\_percent = []

number\_of\_outliers =[]

for n in np.arange(1,100):

    dbscan = DBSCAN(min\_samples=n)

    dbscan.fit\_predict(two\_blobs\_outliers)

    #Total outliers

    number\_of\_outliers.append(np.sum(dbscan.labels\_ == -1))

    #Percentage of points classified as outliers

    perc\_outliers = 100 \* np.sum(dbscan.labels\_ == -1) / len(dbscan.labels\_)

    outlier\_percent.append(perc\_outliers)

**Association learning**

**Apriori**

* In simple words,we weave connections like the following: people who liked product A might also like product B.
* This is divided into 3 parts:
* **Support :** This is a measure of how popular the item is. It is measured by the proportion of transactions in which the item occurs . If we are considering a movie recommendation system then

**support(M) = (number of user watchlists containing movie M)/(total number of user watchlists)**

* **Confidence(X->Y) :** This is a measure of how likely that item Y is purchased when item X is purchased. Extending our movie recommendation example,

**confidence(movie M -> movie N) = (number of user watchlists containing movies M and N)/(number of user watchlists containing movie M)**

* **Lift :** This is how likely item Y is purchased when item X is purchased while controlling how popular item Y is .For the movie recommendation system:

**Lift(M->N) = confidence(M->N)/support(N)**

A lift value greater than 1 means item Y will be purchased when item X is purchased whereas a value less than 1 means that is not the case.

**Algorithm:**

**Step 1:** Set a minimum support and confidence(threshold value)

**Step 2:**  Take alll the subsets in transactions having higher support than minimum support

**Step 3:**  Take all the rules of these subsets having higher confidence than minimum confidence

**Step 4:** Sort the rules by decreasing lift

**Code and implementation**

**Importing libraries**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

df = pd.read\_csv('Market\_Basket\_Optimisation.csv',header=None)

* The apriori function however expects the inputs to be a list of transactions and not a dataframe. hence we convert it to that format
* The inputs must also consist primarily of string datatype.

The following code converts the input data into a list of lists:

**My approach**

transactions\_cst = []

for customer in range(np.shape(df)[0]):

    transactions\_cst.append(list(df.iloc[customer,:]))

**Their approach**

transactions = []

for customer in range(np.shape(df)[0]):

    transactions.append([str(df.values[customer,item]) for item in range(np.shape(df)[1])])

**Training the Apriori model on the dataset**

from apyori import apriori

rules = apriori(transactions=transactions,min\_support = 0.003,min\_confidence = 0.2,min\_lift = 3,min\_length = 2,max\_length = 2)

**Hyperparameters**

* **transcations** : it is the dataset containing the user transactions(or items bought in this case) fed as a list of lists
* **min\_support** : The threshold value of support for the rule. Calculation here is as follows . By common sense, we are considering only those items that appear in atleast 3 transactions a day(this is totally upto us) which makes it 21 times a week . Since we have a total of 7605 transcations we are setting a threshold of 21/7605 ~ 0.003
* **min\_confidence** = For this we use rule of the thumb. In R , this is set by default to 0.8 but that value is found to be too high when used in this case
* **min\_lift** = Again, a matter of trying out values and arriving at the optimal one. Higher the value, better it is
* **min\_length**, **max\_length** : These are the amount of items we want the rule to include. By setting this to 2 here , we will be obatining pairs of products that were brought together. If it were 3, then we would have obtained three products whose consumptions are closely connected.(think of a buy one get two free sale)

**Visualizing the results**

**Displaying the first results coming directly from the output of the apriori function**

results = list(rules)

**Putting the results well organised into a Pandas Dataframe**

def inspect(results):

    lhs         = [tuple(result[2][0][0])[0] for result in results]

    rhs         = [tuple(result[2][0][1])[0] for result in results]

    supports    = [result[1] for result in results]

    confidences = [result[2][0][2] for result in results]

    lifts       = [result[2][0][3] for result in results]

    return list(zip(lhs, rhs, supports, confidences, lifts))

resultsinDataFrame = pd.DataFrame(inspect(results), columns = ['Left Hand Side', 'Right Hand Side', 'Support', 'Confidence', 'Lift'])

**Sorting based on the lift values**

resultsinDataFrame.nlargest(10,columns='Lift')

**Eclat**

* In Eclat model, we only have the support factor. Here however, the support values won’t be calculated for individual items but instead for a set of items.

**Algorithm**

* Set a minimum support.
* Take all subsets in transactions having higher support than minimum support
* Sort these subsets by decreasing support

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

df = pd.read\_csv('Market\_Basket\_Optimisation.csv',header=None)

**Data Preprocessing**

transactions = []

for customer in range(np.shape(df)[0]):

    transactions.append([str(df.values[customer,item]) for item in range(np.shape(df)[1])])

**Training the model**

from apyori import apriori

rules = apriori(transactions=transactions,min\_support = 0.003,min\_confidence = 0.2,min\_lift = 3,min\_length = 2,max\_length = 3)

results = list(rules)

**The only difference in the following snippet is that we only use support and hence eliminate confidence and lifts.**

def inspect(results):

    lhs         = [tuple(result[2][0][0])[0] for result in results]

    rhs         = [tuple(result[2][0][1])[0] for result in results]

    supports    = [result[1] for result in results]

    return list(zip(lhs, rhs, supports))

resultsinDataFrame = pd.DataFrame(inspect(results), columns = ['Product 1', 'Product 2', 'Support'])

resultsinDataFrame.nlargest(10,columns='Support')