Group B
ASSIGNMENT
NO: 4

Title: Implement Gradient Descent Algorithm to find the local minima of a function. For example, find the local minima of the function $y=(x+3)^2$ starting from the point $x=2$.

Objective: Students should learn to implement Gradient Descent Algorithm.

Prerequisite:

1. Basic of Python Language
2. Concept of Gradient Descent Algorithm

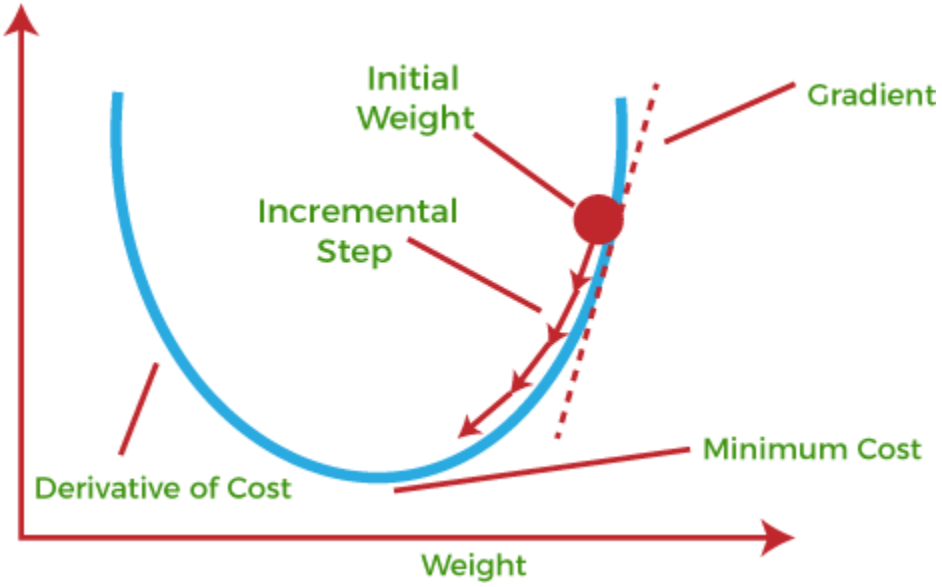
Theory:

Gradient descent was initially discovered by "Augustin-Louis Cauchy" in mid of 18th century. Gradient Descent is defined as one of the most commonly used iterative optimization algorithms of machine learning to train the machine learning and deep learning models. It helps in finding the local minimum of a function.

The best way to define the local minimum or local maximum of a function using gradient descent is as follows:

If we move towards a negative gradient or away from the gradient of the function at the current point, it will give the local minimum of that function.

Whenever we move towards a positive gradient or towards the gradient of the function at the current point, we will get the local maximum of that function.



This entire procedure is known as Gradient Ascent, which is also known as steepest descent. The main objective of using a gradient descent algorithm is to minimize the cost function using iteration.

Types of Gradient Descent

Based on the error in various training models, the Gradient Descent learning algorithm can be divided into **Batch gradient descent, stochastic gradient descent, and mini-batch gradient descent.** Let's understand these different types of gradient descent:

1. Batch Gradient Descent:

Batch gradient descent (BGD) is used to find the error for each point in the training set and update the model after evaluating all training examples. This procedure is known as the training epoch. In simple words, it is a greedy approach where we have to sum over all examples for each update.

Advantages of Batch gradient descent:

- It produces less noise in comparison to other gradient descent. o It produces stable gradient descent convergence.
- It is Computationally efficient as all resources are used for all training samples.

2. Stochastic gradient descent

Stochastic gradient descent (SGD) is a type of gradient descent that runs one training example per iteration. Or in other words, it processes a training epoch for each example within a dataset and updates each training example's parameters one at a time. As it requires only one training example at a time, hence it is easier to store in allocated memory. However, it shows some computational efficiency losses in comparison to batch gradient systems as it shows frequent updates that require more detail and speed. Further, due to frequent updates, it is also treated as a noisy gradient. However, sometimes it can be helpful in finding the global minimum and also escaping the local minimum.

Advantages of Stochastic gradient descent:

In Stochastic gradient descent (SGD), learning happens on every example, and it consists of a few advantages over other gradient descent.

- It is easier to allocate in desired memory. o It is relatively fast to compute than batch gradient descent.
- It is more efficient for large datasets.

3. MiniBatch Gradient Descent:

Mini Batch gradient descent is the combination of both batch gradient descent and stochastic gradient descent. It divides the training datasets into small batch sizes then performs the updates on those batches separately. Splitting training datasets into smaller batches make a balance to maintain the computational efficiency of batch gradient descent and speed of stochastic gradient descent. Hence, we can achieve a special type of gradient descent with higher computational efficiency and less noisy gradient descent.

Advantages of Mini Batch gradient descent:

- It is easier to fit in allocated memory. ○ It is computationally efficient.
- It produces stable gradient descent convergence.

Here is the basic algorithm for Gradient Descent:

Gradient Descent

Algorithm Input:

- Initial parameters θ (often initialized randomly)
- Learning rate α (controls the step size)
- Cost function J(θ) (a function that you want to minimize)
- Convergence threshold ε (a small value that determines when to stop)

Output:

- Optimized parameters θ that minimize the cost function

Procedure:

1. Initialize parameters: Set θ to an initial value, often randomly. Initialize a variable to keep track of the previous cost, J_prev, with a large value (e.g., +∞).
2. Loop until convergence: a. Compute the gradient of the cost function with respect to the parameters: ∇J(θ). b. Update the parameters θ: θ = θ - α * ∇J(θ) This update is performed for each parameter θ_i: θ_i = θ_i - α * ∂J(θ) / ∂θ_i
3. Calculate the new cost: J_new = J(θ).
4. Check for convergence: If |J_new - J_prev| < ε, where ε is the convergence threshold, stop the algorithm. The algorithm has converged, and θ is the optimized parameter set. Otherwise, set J_prev = J_new and go back to step 2.

Hyperparameters:

- Learning Rate (α): The learning rate controls the step size in each update. Choosing an appropriate learning rate is crucial for the convergence and stability of the algorithm. It may require some experimentation.
- Convergence Threshold (ε): The convergence threshold determines when to stop the algorithm. Setting it too low may result in longer training times, while setting it too high may result in suboptimal solutions.

Additional Considerations:

- Gradient Descent can be sensitive to the choice of the learning rate, and various techniques like learning rate schedules or adaptive learning rates (e.g., Adam, RMSprop) can be used to improve convergence.
- Mini-batch Gradient Descent and Stochastic Gradient Descent are variants of this algorithm that use subsets of the training data in each iteration, which can be more computationally efficient.
- Regularization terms can be added to the cost function to prevent overfitting.
- Make sure the cost function is differentiable with respect to the parameters θ for this algorithm to work.

Gradient Descent is a foundational optimization algorithm and serves as the basis for many advanced optimization techniques used in machine learning and deep learning.

Conclusion:

Hence by this way we have successfully implemented GDA.

Group B

Assignment No : 5

Title of the Assignment: Implement K-Nearest Neighbors algorithm on diabetes.csv dataset. Compute confusion matrix, accuracy, error rate, precision and recall on the given dataset.

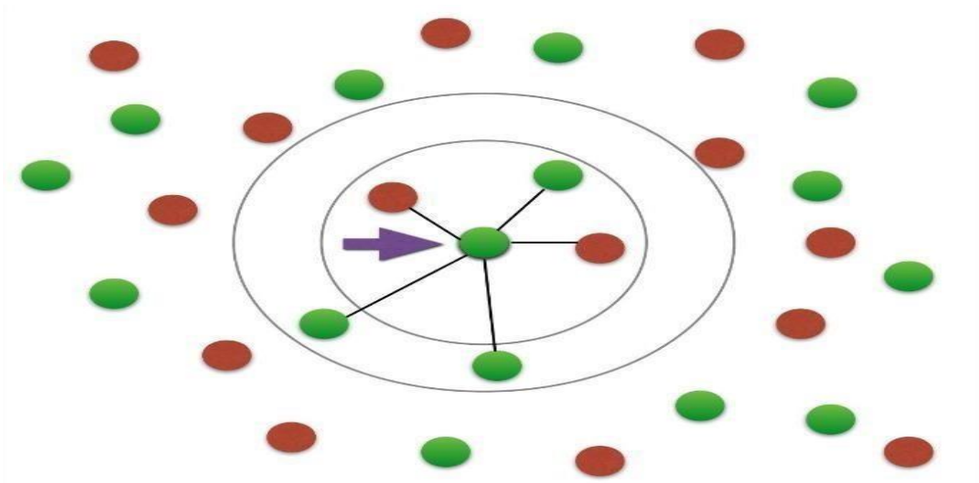
Dataset Description: We will try to build a machine learning model to accurately predict whether or not the patients in the dataset have diabetes or not?
The datasets consists of several medical predictor variables and one target variable, Outcome. Predictor variables includes the number of pregnancies the patient has had, their BMI, insulin level, age, and so on.

Link for Dataset: Diabetes predication system with KNN algorithm | Kaggle

Objective of the Assignment:
Students should be able to preprocess dataset and identify outliers, to check correlation and implement KNN algorithm and random forest classification models. Evaluate them with respective scores like confusion_matrix, accuracy_score, mean_squared_error, r2_score, roc_auc_score, roc_curve etc.

Prerequisite:

1. Basic knowledge of Python
2. Concept of Confusion Matrix
3. Concept of roc_auc curve.
4. Concept of Random Forest and KNN algorithms



k-Nearest-Neighbors (k-NN) is a supervised machine learning model. Supervised learning is when a model learns from data that is already labeled. A supervised learning model takes in a set of input objects and output values. The model then trains on that data to learn how to map the inputs to the desired output so it can learn to make predictions on unseen data.

k-NN models work by taking a data point and looking at the _k‘ closest labeled data points. The data point is then assigned the label of the majority of the _k‘ closest points.
For example, if k = 5, and 3 of points are _green‘ and 2 are _red‘, then the data point in question would be labeled _green‘, since _green‘ is the majority (as shown in the above graph).
Scikit-learn is a machine learning library for Python. In this tutorial, we will build a k-NN model using Scikit-learn to predict whether or not a patient has diabetes. Reading in the training data

For our k-NN model, the first step is to read in the data we will use as input. For this example, we are using the diabetes dataset. To start, we will use Pandas to read in the data. I will not go into detail on Pandas, but it is a library you should become familiar with if you‘re looking to dive further into data science and machine learning.

```
import pandas as pd#read in the data using pandas
df = pd.read_csv(_data/diabetes_data.csv‘)#check data has been read in properly df.head()
```

	pregnancies	glucose	diastolic	triceps	insulin	bmi	dpf	age	diabetes
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Next, let‘s see how much data we have. We will call the _shape‘ function on our dataframe to see how many rows and columns there are in our data. The rows indicate the number of patients and the columns indicate the number of features (age, weight, etc.) in the dataset for each patient.


```
#check number of rows and columns in dataset df.shape

Op → (768,9)
```

We can see that we have 768 rows of data (potential diabetes patients) and 9 columns (8 input features and 1 target output).
Split up the dataset into inputs and targets

Now let‘s split up our dataset into inputs (X) and our target (y). Our input will be every column except _diabetes‘ because _diabetes‘ is what we will be attempting to predict. Therefore, _diabetes‘ will be our target.

We will use pandas _drop‘ function to drop the column _diabetes‘ from our dataframe and store it in the variable _X‘. This will be our input.

```
#create a dataframe with all training data except the target column
X = df.drop(columns=[_diabetes‘])#check that the target variable has been removed
X.head()
```

	pregnancies	glucose	diastolic	triceps	insulin	bmi	dpf	age
0	6	148	72	35	0	33.6	0.627	50
1	1	85	66	29	0	26.6	0.351	31
2	8	183	64	0	0	23.3	0.672	32
3	1	89	66	23	94	28.1	0.167	21
4	0	137	40	35	168	43.1	2.288	33

We will insert the _diabetes‘ column of our dataset into our target variable (y).

```
#separate target values
y = df[_diabetes‘].values#view target values y[0:5]
```



```
array([1, 0, 1, 0, 1])
```

Split the dataset into train and test data

Now we will split the dataset into into training data and testing data. The training data is the data that the model will learn from. The testing data is the data we will use to see how well the model performs on unseen data.

Scikit-learn has a function we can use called _train_test_split‘ that makes it easy for us to split our dataset into training and testing data.

```
from sklearn.model_selection import train_test_split#split dataset into train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1, stratify=y)
```

_train_test_split‘ takes in 5 parameters. The first two parameters are the input and target data we split up earlier. Next, we will set _test_size‘ to 0.2. This means that 20% of all the data will be used for testing, which leaves 80% of the data as training data for the model to learn from. Setting _random_state‘ to 1 ensures that we get the same split each time so we can reproduce our results.

Setting _stratify‘ to y makes our training split represent the proportion of each value in the y variable. For example, in our dataset, if 25% of patients have diabetes and 75% don‘t have diabetes, setting _stratify‘ to y will ensure that the random split has 25% of patients with diabetes and 75% of patients without diabetes.

Building and training the model

Next, we have to build the model. Here is the code:

```
from sklearn.neighbors import KNeighborsClassifier# Create KNN classifier knn = KNeighborsClassifier(n_neighbors = 3)# Fit the classifier to the data knn.fit(X_train,y_train)
```

First, we will create a new k-NN classifier and set _n_neighbors‘ to 3. To recap, this means that if at least 2 out of the 3 nearest points to an new data point are patients without diabetes, then the new data point will be Bharati Vidyapeeth‘s College Of Engineering Lavale Pune. labeled as _no diabetes‘, and vice versa. In other words, a new data point is labeled with by majority from3 the 3 nearest points.

We have set _n_neighbors‘ to 3 as a starting point. We will go into more detail below on how to better select a value for _n_neighbors‘ so that the model can improve its performance.

Next, we need to train the model. In order to train our new model, we will use the _fit‘ function and pass in our training data as parameters to fit our model to the training data.

Testing the model

Once the model is trained, we can use the _predict‘ function on our model to make predictions on our test data. As seen when inspecting _y‘ earlier, 0 indicates that the patient does not have diabetes and 1 indicates that the patient does have diabetes. To save space, we will only show print the first 5 predictions of our test set.

```
#sl                                                    ta
kn array([0, 0, 0, 0, 1])
```

We can see that the model predicted _no diabetes‘ for the first 4 patients in the test set and _has diabetes‘ for the 5th patient.

Now let‘s see how our accurate our model is on the full test set. To do this, we will use the _score‘ function and pass in our test input and target data to see how well our model predictions match up to the actual results.

```
#check accuracy of our model on the test data
knn.score(X_test, y_test)
```

```
0.6688311688311688
```

Our model has an accuracy of approximately 66.88%. It‘s a good start, but we will see how we can increase model performance below.

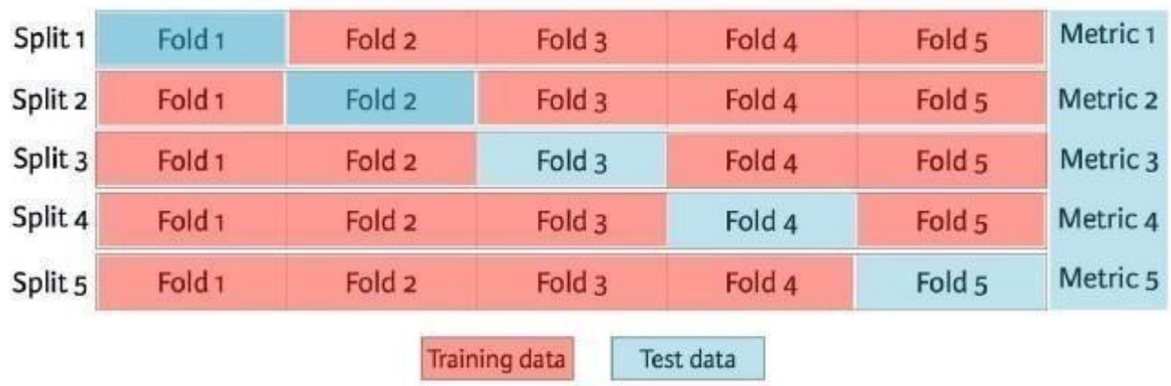
Congrats! You have now built an amazing k-NN model!

k-Fold Cross-Validation

Cross-validation is when the dataset is randomly split up into _k‘ groups. One of the groups is used as the test set

and the rest are used as the training set. The model is trained on the training set and scored on the test set. Then the process is repeated until each unique group as been used as the test set.

For example, for 5-fold cross validation, the dataset would be split into 5 groups, and the model would be trained and tested 5 separate times so each group would get a chance to be the test set. This can be seen in the graph below.



5-fold cross validation (image credit)

The train-test-split method we used in earlier is called _holdout‘. Cross-validation is better than using the holdout method because the holdout method score is dependent on how the data is split into train and test sets. Cross-validation gives the model an opportunity to test on multiple splits so we can get a better idea on how the model will perform on unseen data.

In order to train and test our model using cross-validation, we will use the _cross_val_score‘ function with a cross-validation value of 5. _cross_val_score‘ takes in our k-NN model and our data as parameters. Then it splits our data into 5 groups and fits and scores our data 5 seperate times, recording the accuracy score in an array each time. We will save the accuracy scores in the _cv_scores‘ variable.

To find the average of the 5 scores, we will use numpy‘s mean function, passing in _cv_score‘. Numpy is a useful math library in Python.

```
from sklearn.model_selection import cross_val_score
import numpy as np#create a new KNN model
knn_cv = KNeighborsClassifier(n_neighbors=3)#train model with cv of 5 cv_scores =
cross_val_score(knn_cv, X, y, cv=5)#print each cv score (accuracy) and average them
print(cv_scores)
print(_cv_scores mean:{}‘.format(np.mean(cv_scores)))
```

```
[ 0.68181818  0.69480519  0.75324675  0.75163399  0.68627451]
cv_scores mean:0.7135557253204311
```


Using cross-validation, our mean score is about 71.36%. This is a more accurate representation of how our model will perform on unseen data than our earlier testing using the holdout method.

Hypertuning model parameters using GridSearchCV

When built our initial k-NN model, we set the parameter _n_neighbors‘ to 3 as a starting point with no real logic behind that choice.

Hypertuning parameters is when you go through a process to find the optimal parameters for yourmodel to improve accuracy. In our case, we will use GridSearchCV to find the optimal value for _n_neighbors‘.

GridSearchCV works by training our model multiple times on a range of parameters that we specify. That way, we can test our model with each parameter and figure out the optimal values to get the best accuracy results.

For our model, we will specify a range of values for _n_neighbors‘ in order to see which value works best for our model. To do this, we will create a dictionary, setting _n_neighbors‘ as the key and using numpy to create an array of values from 1 to 24.

Our new model using grid search will take in a new k-NN classifier, our param_grid and a cross- validation value of 5 in order to find the optimal value for _n_neighbors‘.

```
from sklearn.model_selection import GridSearchCV#create new a knn model knn2 =
KNeighborsClassifier()#create a dictionary of all values we want to test for n_neighbors
param_grid = {_n_neighbors‘: np.arange(1, 25)}#use gridsearch to test all values for
n_neighbors knn_gscv = GridSearchCV(knn2, param_grid, cv=5)#fit model to data
knn_gscv.fit(X, y)
```

After training, we can check which of our values for _n_neighbors‘ that we tested performed the best. To do this, we will call _best params ‘ on our model.

```
#check top performing n_neighbors value
knn_gscv.best_params_
```

```
{'n_neighbors': 14}
```

We can see that 14 is the optimal value for _n_neighbors‘. We can use the _best_score_‘ function to check the accuracy of our model when _n_neighbors‘ is 14. _best_score_‘ outputs the mean accuracy of the scores obtained through cross-validation.


```
#check mean score for the top performing value of n_neighbors
knn_gscv.best_score_
```

```
0.7578125
```

By using grid search to find the optimal parameter for our model, we have improved our model accuracy by over 4%!

Code :- https://www.kaggle.com/code/shrutimechlearn/step-by-step-diabetes-classification-knn-detailed

Conclusion:

In this way we build a a neural network-based classifier that can determine whether they willleave or not in the next 6 months

Group B

Assignment No : 6

Title of the Assignment: Implement K-Means clustering/ hierarchical clustering on sales_data_sample.csv dataset. Determine the number of clusters using the elbow method.

Dataset Description: The data includes the following features:

1. Customer ID
2. Customer Gender
3. Customer Age
4. Annual Income of the customer (in Thousand Dollars)
5. Spending score of the customer (based on customer behavior and spending nature)

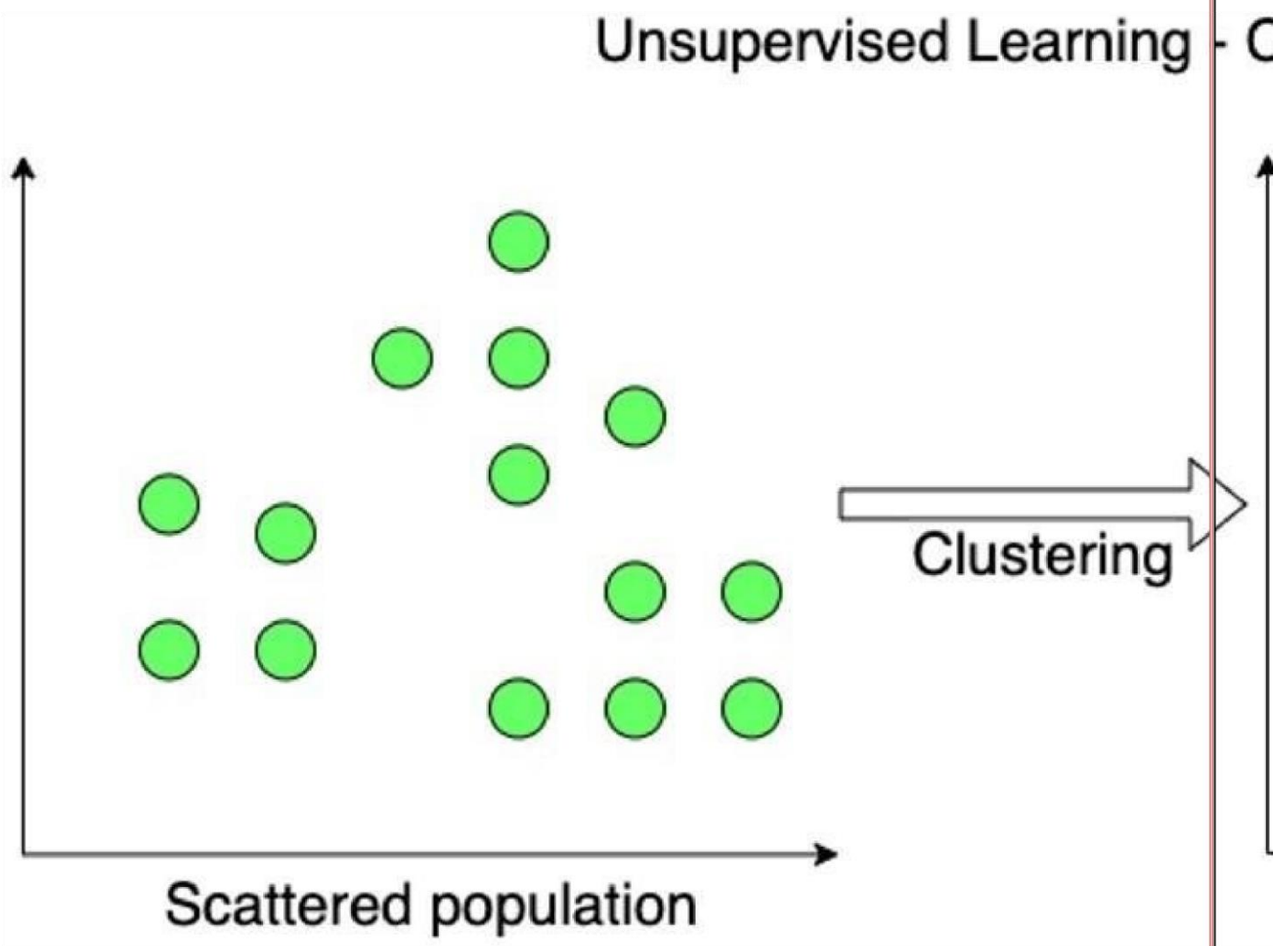
Objective of the Assignment:

Students should able to understand how to use unsupervised learning to segment different-different clusters or groups and used to them to train your model to predict future things.

Prerequisite:

1. Knowledge of Python
2. Unsupervised learning
3. Clustering
4. Elbow method

Clustering algorithms try to find natural clusters in data, the various aspects of how the algorithms to cluster data can be tuned and modified. Clustering is based on the principle that items within the same cluster must be similar to each other. The data is grouped in such a way that related elements are close to each other.



Diverse and different types of data are subdivided into smaller groups.

Uses of Clustering

Marketing:

In the field of marketing, clustering can be used to identify various customer groups with existing customer data. Based on that, customers can be provided with discounts, offers, promo codes etc.

Real Estate:

Clustering can be used to understand and divide various property locations based on value and importance. Clustering algorithms can process through the data and identify various groups of property on the basis of probable price.

BookStore and Library management:

Libraries and Bookstores can use Clustering to better manage the book database. With proper book ordering, better operations can be implemented.

Document Analysis:

Often, we need to group together various research texts and documents according to similarity. And in such cases, we don‘t have any labels. Manually labelling large amounts of data is also not possible. Using clustering, the algorithm can process the text and group it into different themes.

These are some of the interesting use cases of clustering.

K-Means Clustering

K-Means clustering is an unsupervised machine learning algorithm that divides the given data into the given number of clusters. Here, the ―K‖ is the given number of predefined clusters, that need to be created.

It is a centroid based algorithm in which each cluster is associated with a centroid. The main idea is to reduce the distance between the data points and their respective cluster centroid.

The algorithm takes raw unlabelled data as an input and divides the dataset into clusters and the process is repeated until the best clusters are found.

K-Means is very easy and simple to implement. It is highly scalable, can be applied to both small and large datasets. There is, however, a problem with choosing the number of clusters or K. Also, with the increase in dimensions, stability decreases. But, overall K Means is a simple and robust algorithm that makes clustering very easy.

```
#Importing the necessary librariesimport
numpy as np import pandas as pd import
matplotlib.pyplot as pltimport seaborn as
sns from mpl_toolkits.mplot3d import
Axes3D
%matplotlib inline
```

The necessary libraries are imported.

```
#Reading the excel file data=pd.read_excel("Mall_Customers.xlsx")
```

The data is read. I will share a link to the entire code and excel data at the end of the article.

TheBharati data Vidyapeeth‘s has 200 entries, College that Of isEngineering data from 200 Lavale customers. Pune.


```
data.head()
```

So let us have a look at the data.

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

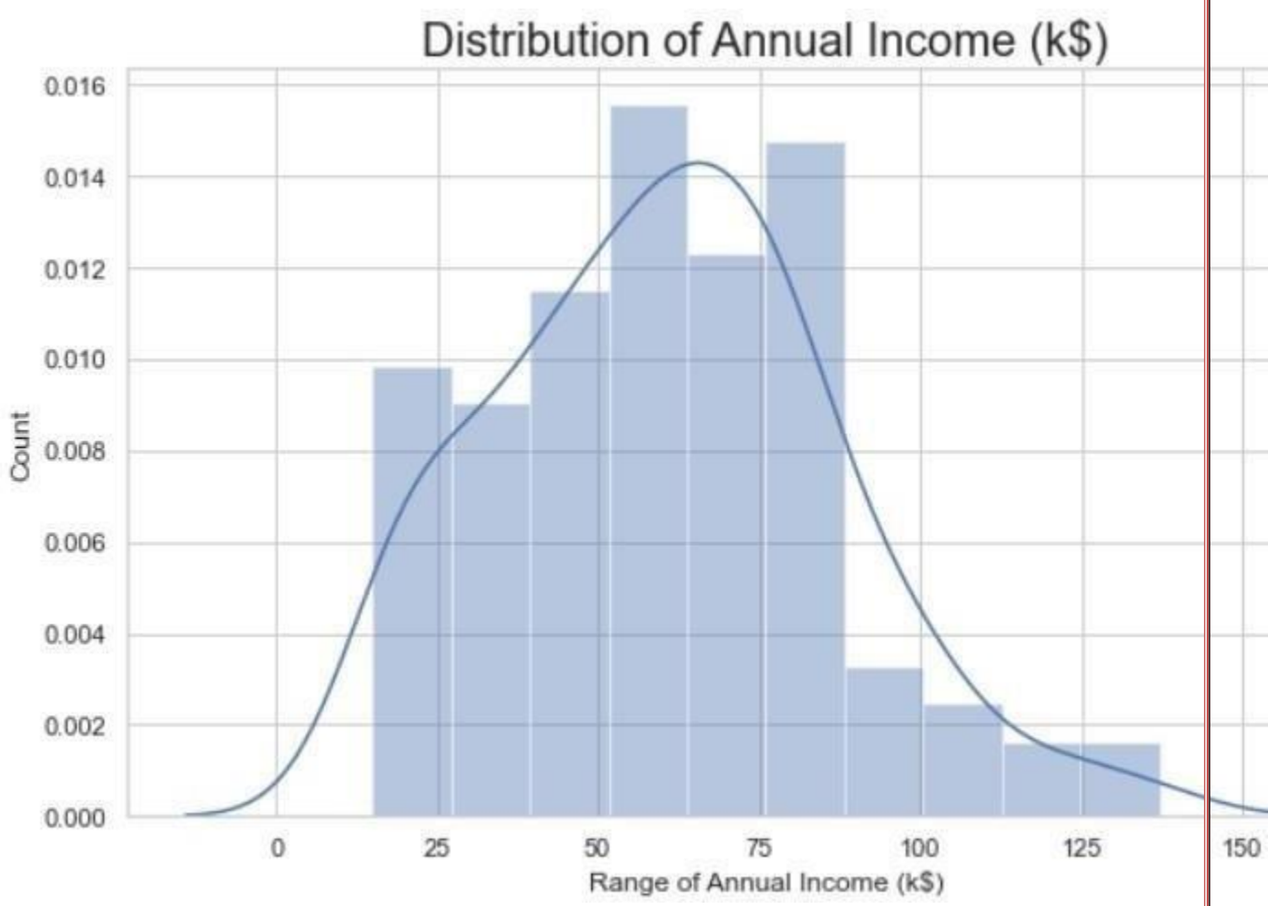
data.corr()

	CustomerID	Age	Annual Income (k$)	Spending Score (
CustomerID	1.000000	-0.026763	0.977548	0.013835
Age	-0.026763	1.000000	-0.012398	-0.327227
Annual Income (k$)	0.977548	-0.012398	1.000000	0.009903
Spending Score (1-100)	0.013835	-0.327227	0.009903	1.000000

The data seems to be interesting. Let us look at the data distribution.

Annual Income Distribution:

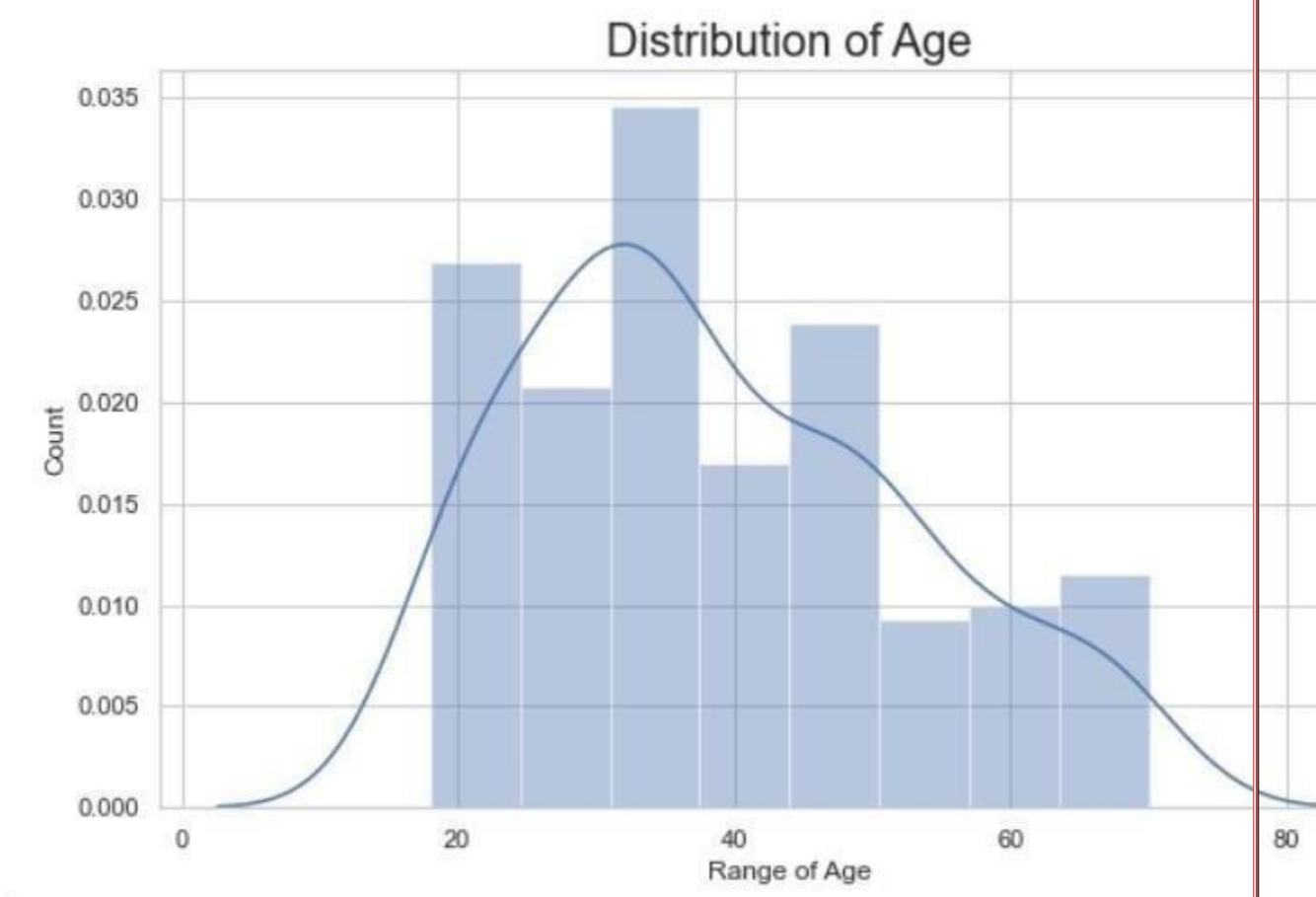
```
#Distribution of Annnual Income
plt.figure(figsize=(10, 6)) sns.set(style = 'whitegrid')
sns.distplot(data['Annual Income (k$)'])
plt.title('Distribution of Annual Income (k$)', fontsize = 20)plt.xlabel('Range of Annual
Income (k$)')
plt.ylabel('Count')
```



Most of the annual income falls between 50K to 85K.

Age Distribution:

```
#Distribution of age
plt.figure(figsize=(10, 6)) sns.set(style
= 'whitegrid')sns.distplot(data['Age'])
plt.title('Distribution of Age', fontsize = 20)plt.xlabel('Range of
Age')
plt.ylabel('Count')
```



There are customers of a wide variety of ages.

Spending Score Distribution:

#Distribution of spending score

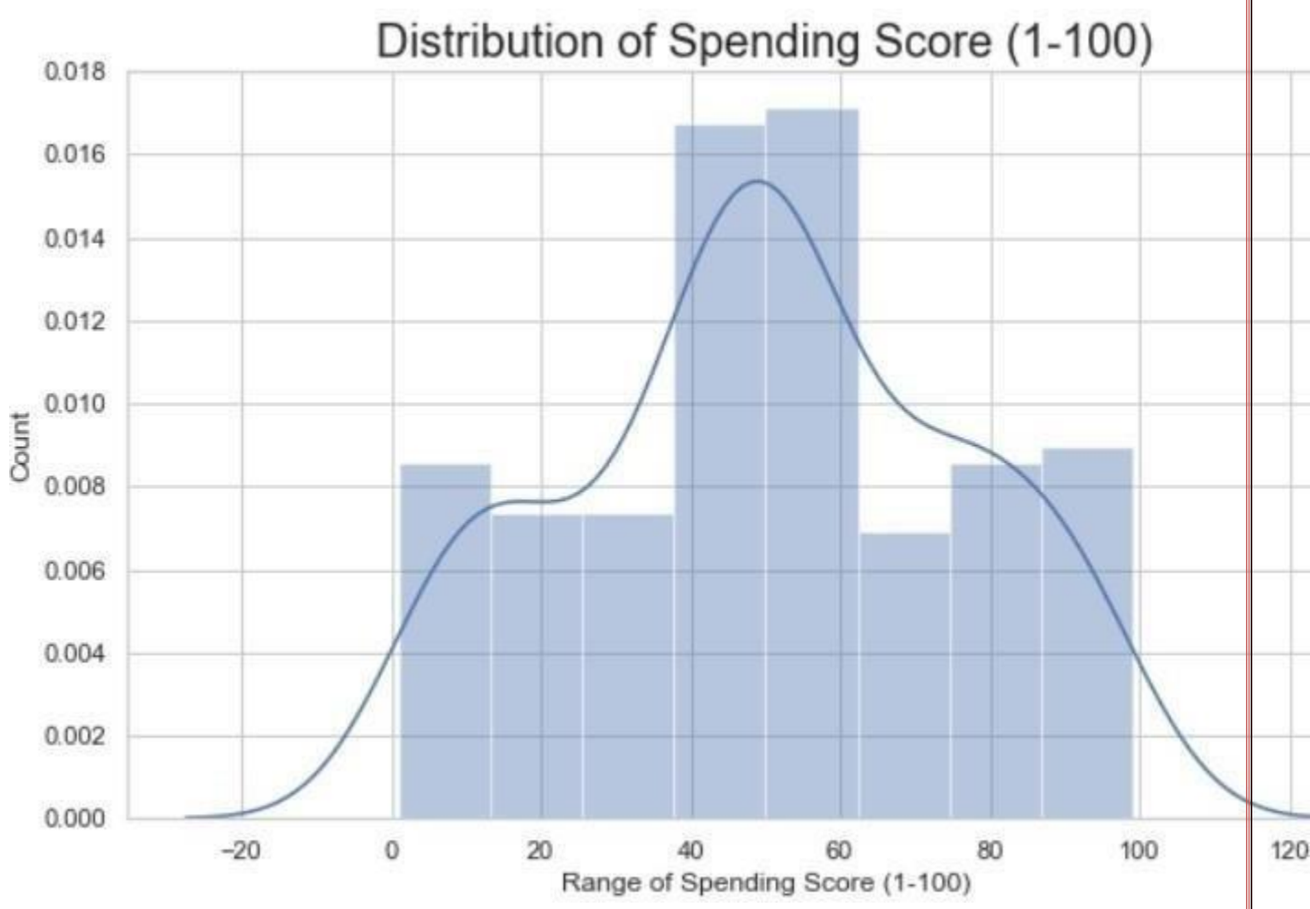
plt.figure(figsize=(10, 6)) sns.set(style =

'whitegrid')

sns.distplot(data['Spending Score (1-100)']) plt.title('Distribution of Spending Score (1-

100)', fontsize = 20)plt.xlabel('Range of Spending Score (1-100)')

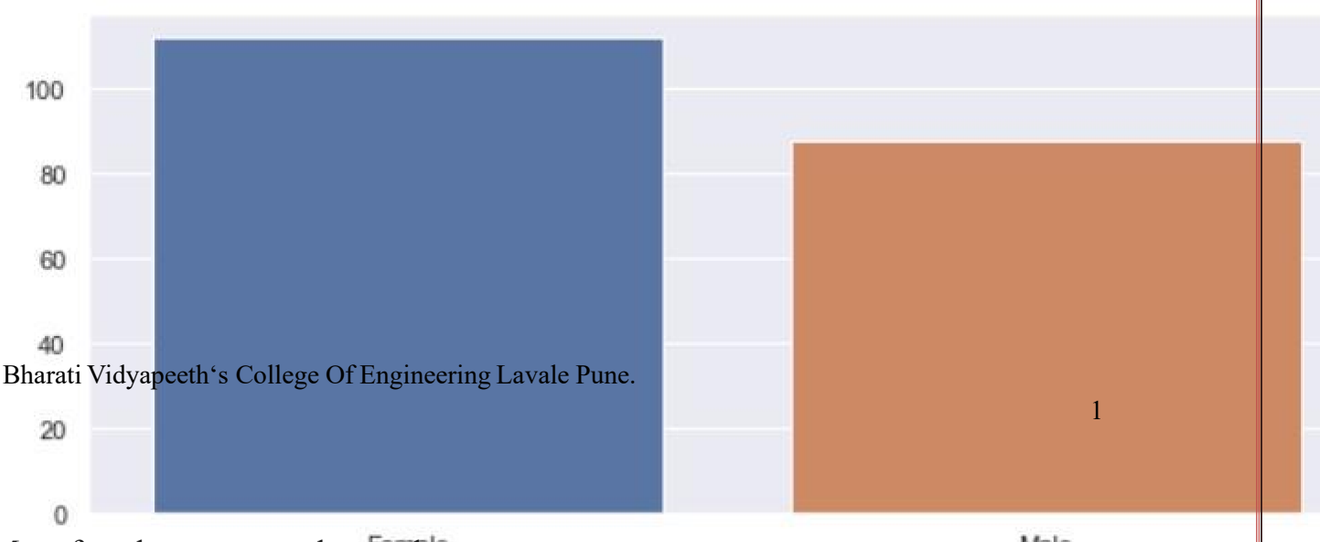
plt.ylabel('Count')



The maximum spending score is in the range of 40 to 60.

Gender Analysis:

```
genders = data.Gender.value_counts() sns.set_style("darkgrid")
plt.figure(figsize=(10,4)) sns.barplot(x=genders.index,
y=genders.values)plt.show()
```



More female customers than male.

I have made more visualizations. Do have a look at the GitHub link at the end to understand the data analysis and overall data exploration.

Clustering based on 2 features

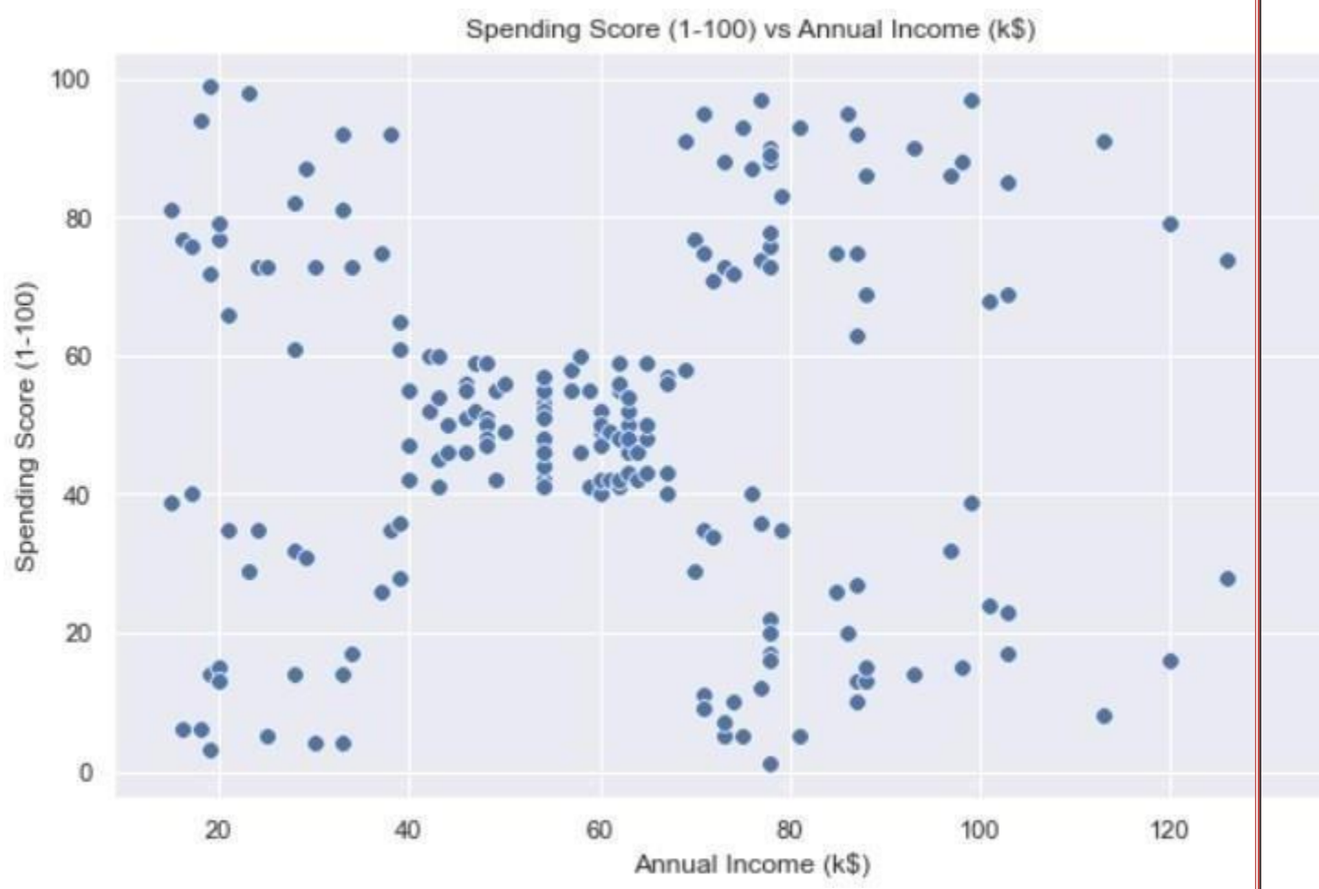
First, we work with two features only, annual income and spending score.

```
#We take just the Annual Income and Spending score  df1=data[["CustomerID","Gender","Age","Annual Income
(k$)","Spending Score (1-100)"]]X=df1[["Annual Income (k$)","Spending Score (1-100)"]]
#The input data
X.head()
```

	Annual Income (k$)	Spending Score (1-100)
0	15	39
1	15	81
2	16	6
3	16	77
4	17	40

```
#Scatterplot of the input data
plt.figure(figsize=(10,6))
sns.scatterplot(x = 'Annual Income (k$)',y = 'Spending Score (1-100)',   data = X   ,s = 60
)
plt.xlabel('Annual Income (k$)') plt.ylabel('Spending
Score (1-100)')
plt.title('Spending Score (1-100) vs Annual Income (k$)')plt.show()
```

The data does seem to hold some patterns.

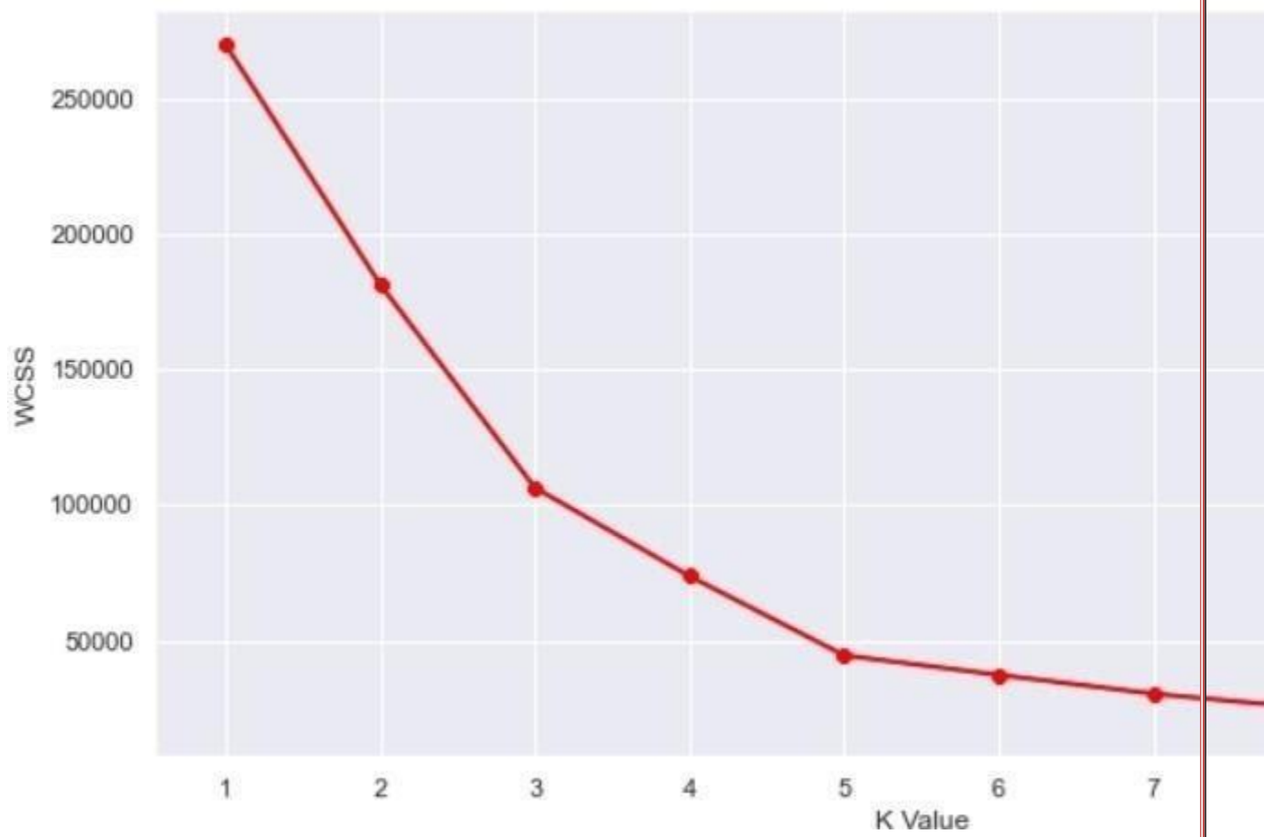



```
#Importing KMeans from sklearn from
sklearn.cluster import KMeans
```

Now we calculate the Within Cluster Sum of Squared Errors (WSS) for different values of k. Next, we choose the k for which WSS first starts to diminish. This value of K gives us the best number of clusters to make from the raw data.

```
wcss=[]
for i in range(1,11):
        km=KMeans(n_clusters=i)km.fit(X)
        wcss.append(km.inertia_)
#The elbow curve plt.figure(figsize=(12,6))
plt.plot(range(1,11),wcss)
plt.plot(range(1,11),wcss, linewidth=2, color="red", marker ="8")plt.xlabel("K Value")
plt.xticks(np.arange(1,11,1))
```


The plot:



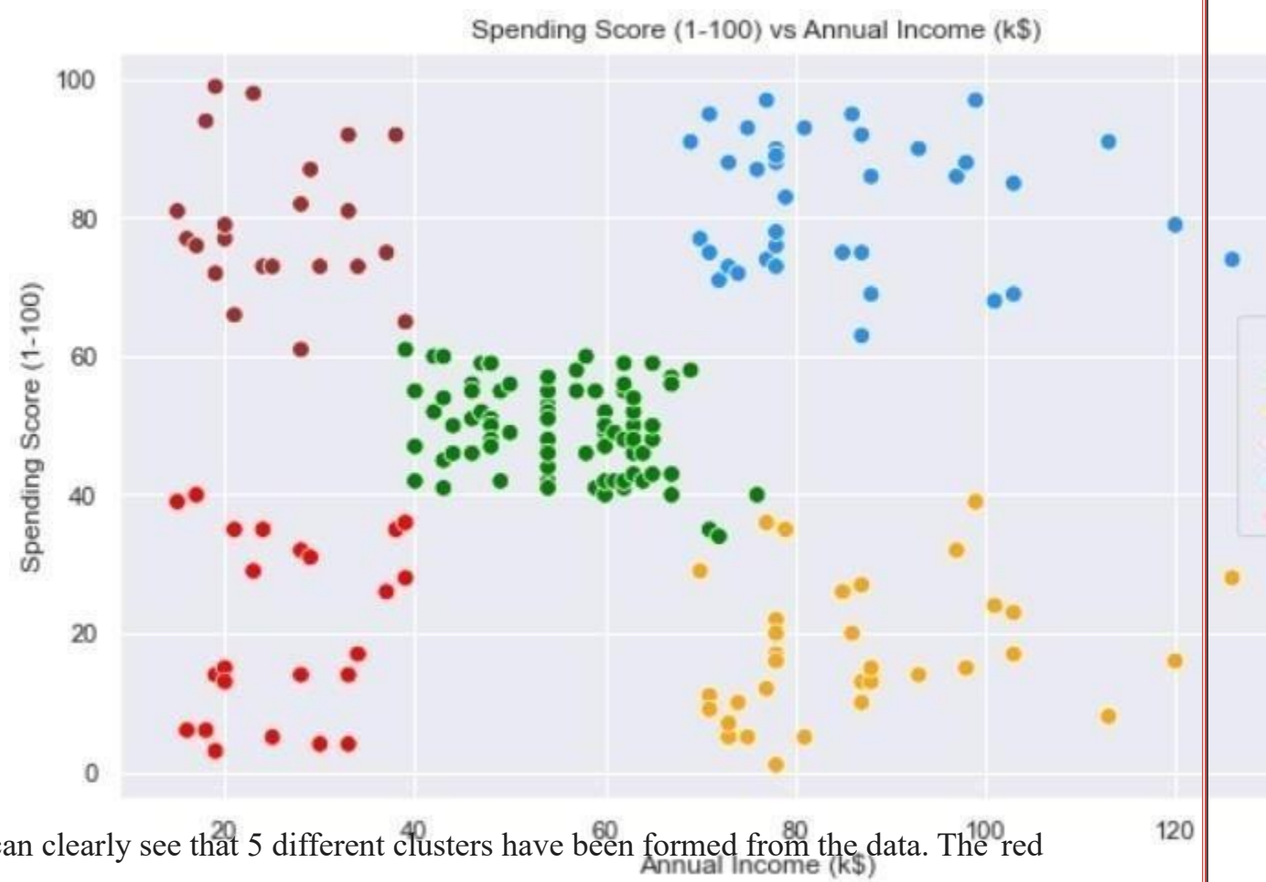
This is known as the elbow graph, the x-axis being the number of clusters, the number of clusters is taken at the elbow joint point. This point is the point where making clusters is most relevant as here the value of WCSS suddenly stops decreasing. Here in the graph, after 5 the drop is minimal, so we take 5 to be the number of clusters.

```
#Taking 5 clusters
km1=KMeans(n_clusters=5)#Fitting the
input data km1.fit(X)
#predicting the labels of the input data
y=km1.predict(X)
#adding the labels to a column named labeldf1["label"] =
y
#The new dataframe with the clustering donedf1.head()
```

The labels added to the data.

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)	label
0	1	Male	19	15	39	4
1	2	Male	21	15	81	2
2	3	Female	20	16	6	4
3	4	Female	23	16	77	2
4	5	Female	31	17	40	4

```
#Scatterplot of the clusters
plt.figure(figsize=(10,6))
sns.scatterplot(x = 'Annual Income (k$)',y = 'Spending Score (1-100)',hue="label",
                palette=['green','orange','brown','dodgerblue','red'],legend='full',data = df1  ,s = 60 )
plt.xlabel('Annual Income (k$)') plt.ylabel('Spending
Score (1-100)')
plt.title('Spending Score (1-100) vs Annual Income (k$)') plt.show()
```



We can clearly see that 5 different clusters have been formed from the data. The red cluster is the customers with the least income and least spending score, similarly, the blue cluster is the customers with the most income and most spending score.

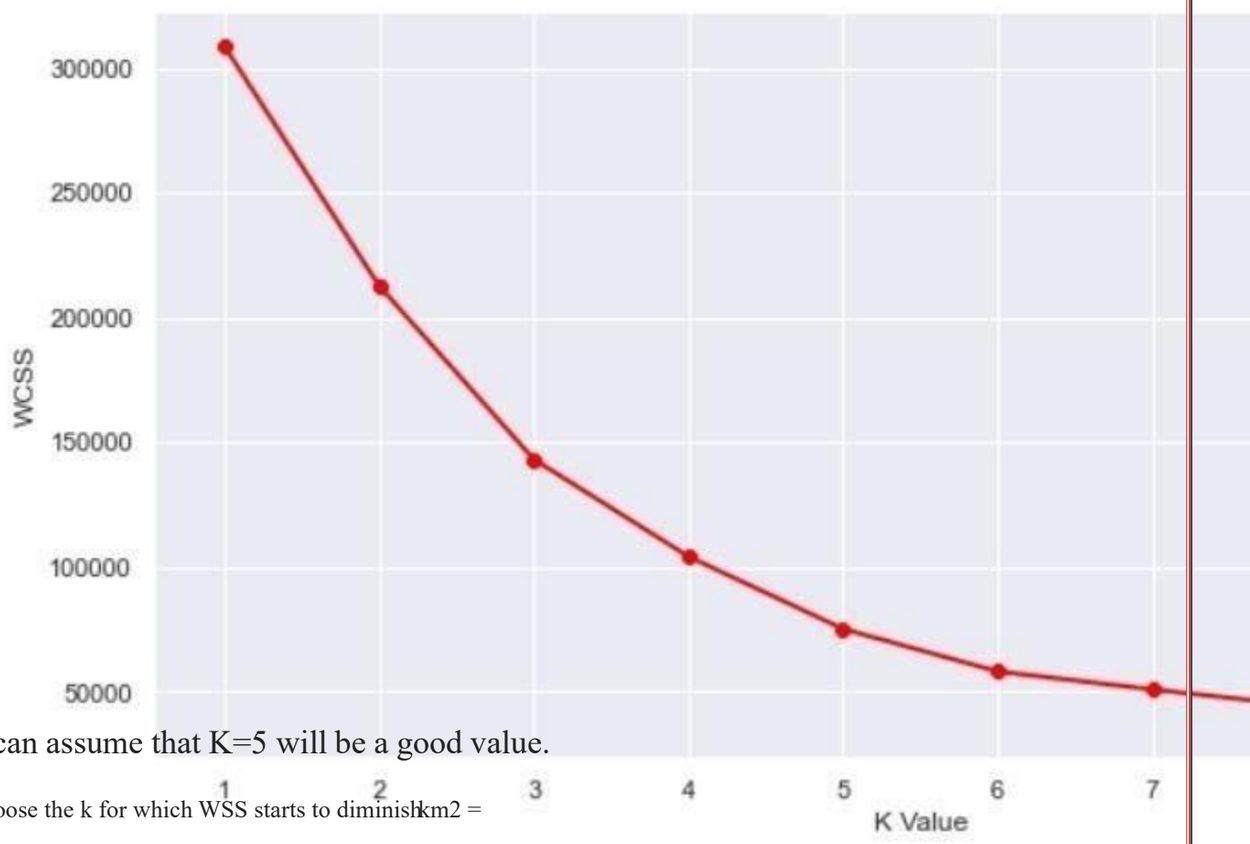
k-Means Clustering on the basis of 3D data

Now, we shall be working on 3 types of data. Apart from the spending score and

annual income of customers, we shall also take in the age of the customers.

```
#Taking the features
X2=df2[["Age","Annual Income (k$)","Spending Score (1-100)"]]
#Now we calculate the Within Cluster Sum of Squared Errors (WSS) for different values of k.
wcss = []
for k in range(1,11):
    kmeans = KMeans(n_clusters=k, init="k-means++").kmeans.fit(X2)
    wcss.append(kmeans.inertia_)
plt.figure(figsize=(12,6))
plt.plot(range(1,11),wcss, linewidth=2, color="red", marker ="8").plt.xlabel("K Value")
plt.xticks(np.arange(1,11,1))
plt.ylabel("WCSS") plt.show()
```

The WCSS curve.



Here can assume that K=5 will be a good value.

```
#We choose the k for which WSS starts to diminish.km2 =
KMeans(n_clusters=5)
y2 = km.fit_predict(X2)
df2["label"] = y2
#The data with labels
df2.head()
```

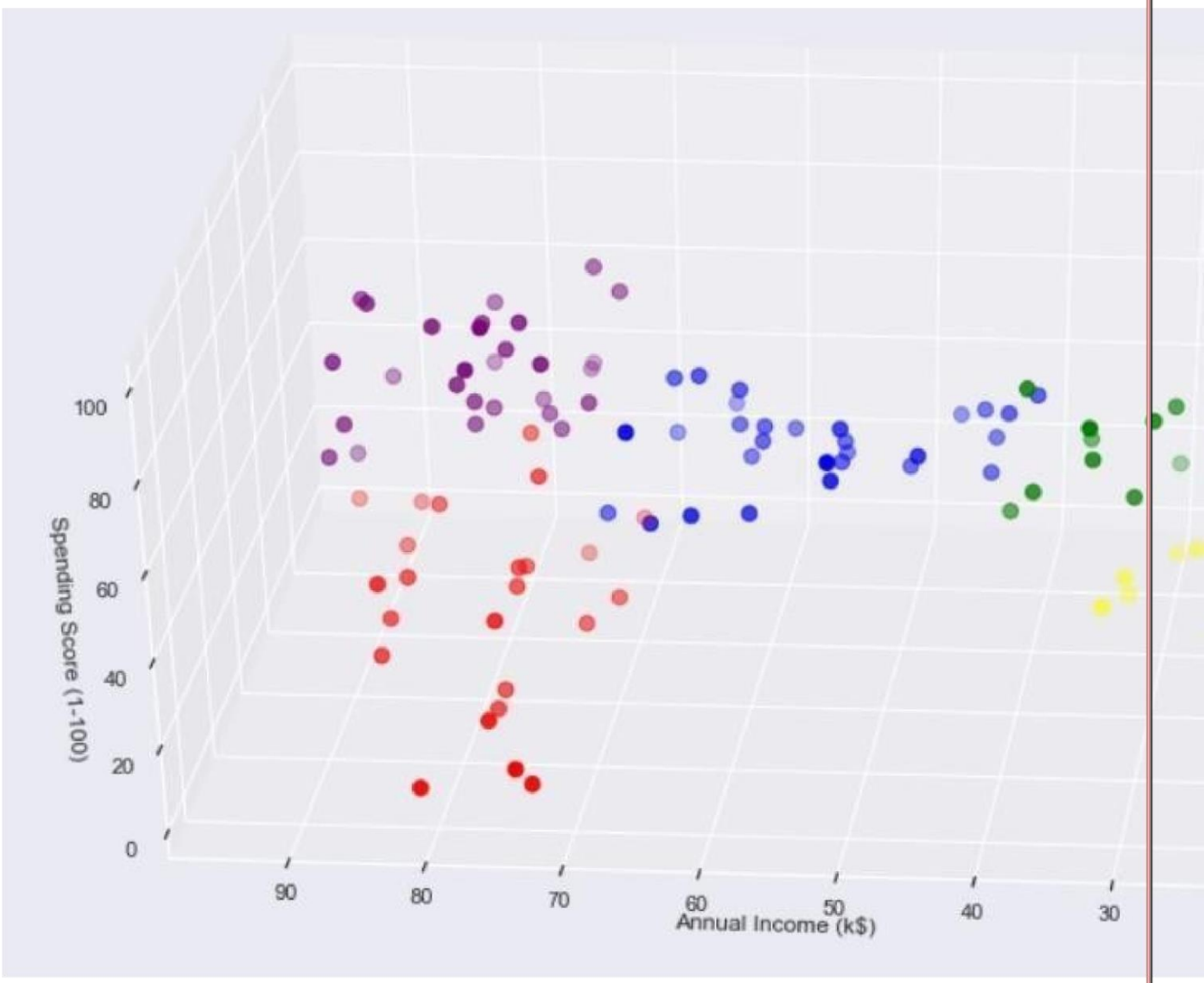
The data:

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)	label
0	1	Male	19	15	39	5
1	2	Male	21	15	81	3
2	3	Female	20	16	6	4
3	4	Female	23	16	77	3
4	5	Female	31	17	40	5

Now we plot it.

```
#3D Plot as we did the clustering on the basis of 3 input features fig =

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

ax = fig.add_subplot(111, projection='3d')

ax.scatter(df2.Age[df2.label == 0], df2["Annual Income (k$)"][df2.label == 0], df2["Spending Score (1-
100)"][df2.label == 0], c='purple', s=60)

ax.scatter(df2.Age[df2.label == 1], df2["Annual Income (k$)"][df2.label == 1], df2["Spending Score (1-
100)"][df2.label == 1], c='red', s=60)

ax.scatter(df2.Age[df2.label == 2], df2["Annual Income (k$)"][df2.label == 2], df2["Spending Score (1-
100)"][df2.label == 2], c='blue', s=60)

ax.scatter(df2.Age[df2.label == 3], df2["Annual Income (k$)"][df2.label == 3], df2["Spending Score (1-
100)"][df2.label == 3], c='green', s=60)

ax.scatter(df2.Age[df2.label == 4], df2["Annual Income (k$)"][df2.label == 4], df2["Spending Score (1-
100)"][df2.label == 4], c='yellow', s=60)

ax.view_init(35, 185) plt.xlabel("Age")

plt.ylabel("Annual Income (k$)")

ax.set_zlabel('Spending Score (1-100)') plt.show()
```

The output:



What we get is a 3D plot. Now, if we want to know the customer IDs, we can do that too.

```
cust1=df2[df2["label"]==1]
print('Number of customer in 1st group=', len(cust1))print('They are -',
cust1["CustomerID"].values) print("                                        ")
cust2=df2[df2["label"]==2]
print('Number of customer in 2nd group=', len(cust2))print('They are -',
cust2["CustomerID"].values) print("                                        ")
cust3=df2[df2["label"]==0]
print('Number of customer in 3rd group=', len(cust3))print('They are -',
cust3["CustomerID"].values) print("                                        ")

print('Number of customer in 4th group=', len(cust4))
print('They are -', cust4["CustomerID"].values) print("                                        ")
cust5=df2[df2["label"]==4]
print('Number of customer in 5th group=', len(cust5))print('They are -',
cust5["CustomerID"].values)
```

```
print("                                                                                                    ")
```

The output we get:

```
Number of customer in 1st group= 24
They are - [129 131 135 137 139 141 145 147 149 151 153 155 157 159 161 163 165 167
169 171 173 175 177 179]

_____

Number of the customer in 2nd group= 29
They are - [ 47 51 55 56 57 60 67 72 77 78 80 82 84 86 90 93 94 97
99 102 105 108 113 118 119 120 122 123 127]

_____

Number of the customer in 3rd group= 28

They are - [124 126 128 130 132 134 136 138 140 142 144 146 148 150 152 154 156 158

160 162 164 166 168 170 172 174 176 178]

_____

Number of the customer in 4th group= 22
They are - [ 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 46]
--------------------------------------------------------------------------------

Number of customer in 5th group= 12 They
are - [ 3 7 9 11 13 15 23 25 31 33 35 37]

_____
```

So, we used K-Means clustering to understand customer data. K-Means is a good clustering algorithm. Almost all the clusters have similar density. It is also fast and efficient in terms of computational cost.

MINI PROJECT 2

Problem Statement: - Build a machine learning model that predicts the type of people who survived the Titanic shipwreck using passenger data (i.e. name, age, gender, socio-economic class,etc.).

Objective: Students should learn to build a machine learning model.

Theory:

Here's a step-by-step guide on how to approach this problem using Python and some popular libraries:

1. **Data Collection and Understanding**:
    - Start by obtaining the Titanic dataset, which contains passenger information and survival labels. You can find datasets on websites like Kaggle.
2. **Data Pre-processing**:
    - Clean the data by handling missing values, outliers, and redundant features.
    - Perform feature engineering to create relevant features or transform existing ones.
    - Encode categorical variables into numerical format using techniques like one-hot encoding.
3. **Data Splitting**:
    - Split your dataset into a training set and a test set. This allows you to evaluate your model's performance on unseen data.
4. **Select a Machine Learning Algorithm**:
    - Choose a classification algorithm suitable for this problem. Common choices include Decision Trees, Random Forests, Logistic Regression, Support Vector Machines, or Gradient Boosting.
5. **Model Training**:
    - Fit your chosen algorithm to the training data. The model learns patterns from the data.
6. **Model Evaluation**:
    - Evaluate your model's performance using metrics like accuracy, precision, recall, F1-score, and the ROC-AUC score. Cross-validation can help in assessing how well the model generalizes to new data.
7. **Hyperparameter Tuning**:
    - Experiment with different hyperparameters to optimize your model's performance. Techniques like grid search or random search can be helpful.
8. **Model Interpretation**:
    - Understand the feature importance or coefficients of your model to interpret how different features affect survival.
9. **Prediction**:
    - Use your trained model to make predictions on new, unseen data or the test set.

10. **Post-processing**:

- You may need to further process the model's output, such as setting a threshold for classification.

Importing the Libraries

```
# linear algebra
import numpy as np

#    data    processing
import pandas as pd

# data visualization
import seaborn as sns
%matplotlib inline
from matplotlib import pyplot as plt
from matplotlib import style

# Algorithms
from    sklearn    import    linear_model    from
sklearn.linear_model   import   LogisticRegression
from           sklearn.ensemble           import
RandomForestClassifier  from  sklearn.linear_model
import Perceptron from sklearn.linear_model import
SGDClassifier     from     sklearn.tree     import
DecisionTreeClassifier   from   sklearn.neighbors
import   KNeighborsClassifier   from   sklearn.svm
import  SVC,  LinearSVC  from  sklearn.naive_bayes
import GaussianNB
```

Getting the Data

```
test_df = pd.read_csv("test.csv")
train_df = pd.read_csv("train.csv")
```

Data Exploration/Analysis

```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId     891 non-null int64
Survived        891 non-null int64
Pclass          891 non-null int64
Name            891 non-null object
Sex             891 non-null object
Age             714 non-null float64
SibSp           891 non-null int64
Parch           891 non-null int64
Ticket          891 non-null object
Fare            891 non-null float64
Cabin           204 non-null object
Embarked        889 non-null object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.6+ KB
```

The training-set has 891 examples and 11 features + the target variable (survived). 2 of the features are floats, 5 are integers and 5 areobjects. Below I have listed the features with a short description:

survival: Survival

```
PassengerId: Unique Id of a passenger.
pclass:    Ticket class
sex:    Sex
Age:    Age in years
sibsp: # of siblings / spouses aboard the Titanic
parch: # of parents / children aboard the Titanic
ticket:  Ticket number
fare:   Passenger  fare
cabin:    Cabin number
embarked:    Port of Embarkationtrain_df.describe()
```

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

Above we can see that **38% out of the training-set survived the Titanic**. We can also see that the passenger ages range from 0.4 to 80. Ontop of that we can already detect some features, that contain missing values, like the „Age" feature.

```
train_df.head(8)
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
				Harris								
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
5	6	0	3	Moran, Mr. James	male	NaN	0	0	330877	8.4583	NaN	Q
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349909	21.0750	NaN	S
8	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27.0	0	2	347742	11.1333	NaN	S

From the table above, we can note a few things. First of all, that we **need to convert a lot of features into numeric** ones later on, so that the machine learning algorithms can process them. Furthermore, we can see that the **features have widely different ranges**, that we will need to

convert into roughly the same scale. We can also spot some more features,that contain missing values (NaN = not a number), that wee need to deal with.

Let's take a more detailed look at what data is actually missing:

```
total = train_df.isnull().sum().sort_values(ascending=False)
percent_1 = train_df.isnull().sum()/train_df.isnull().count()*100
percent_2 = (round(percent_1, 1)).sort_values(ascending=False)
missing_data = pd.concat([total, percent_2], axis=1,
keys=['Total','%'])
missing_data.head(5)
```

	Total	%
Cabin	687	77.1
Age	177	19.9
Embarked	2	0.2
Fare	0	0.0
Ticket	0	0.0

The Embarked feature has only 2 missing values, which can easily be filled. It will be much more tricky, to deal with the „Age" feature, which has177 missing values. The „Cabin" feature needs further investigation, but it looks like that we might want to drop it from the dataset, since 77 % of it are missing.

```
train_df.columns.values
```

```
array(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'], dtype=object)
```

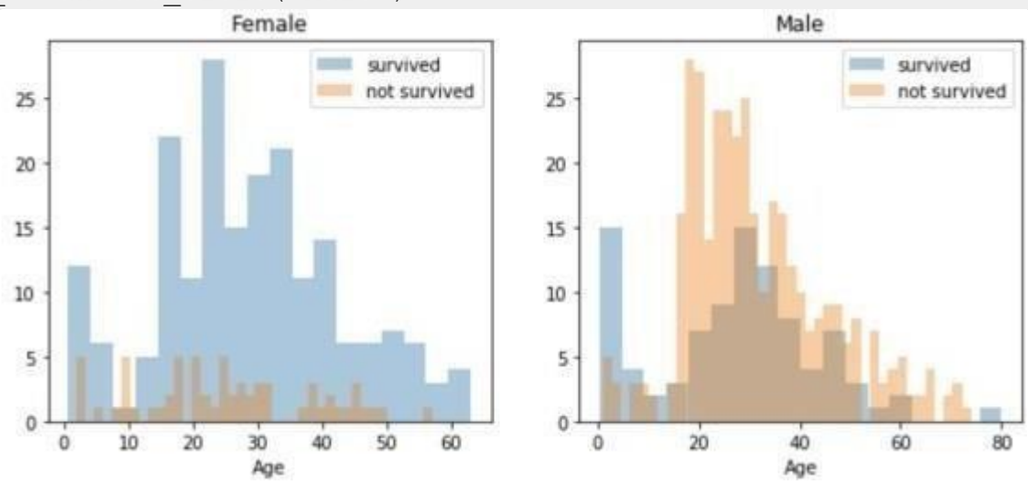
Above you can see the 11 features + the target variable (survived). **Whatfeatures could contribute to a high survival rate ?**

To me it would make sense if everything except „PassengerId", „Ticket" and„Name" would be correlated with a high survival rate.

1. Age and Sex:

```
survived = 'survived'
not_survived = 'not survived'
fig, axes = plt.subplots(nrows=1, ncols=2,figsize=(10,
4))women = train_df[train_df['Sex']=='female']
men = train_df[train_df['Sex']=='male']
ax = sns.distplot(women[women['Survived']==1].Age.dropna(), bins=18,
label = survived, ax = axes[0], kde =False)
ax = sns.distplot(women[women['Survived']==0].Age.dropna(), bins=40,
label = not_survived, ax = axes[0], kde =False)
ax.legend()
ax.set_title('Female')
ax = sns.distplot(men[men['Survived']==1].Age.dropna(), bins=18, label
= survived, ax = axes[1], kde = False)
```

```
ax = sns.distplot(men[men['Survived']==0].Age.dropna(), bins=40,
label
= not_survived, ax = axes[1], kde = False)
ax.legend()
_ = ax.set_title('Male')
```



You can see that men have a high probability of survival when they are between 18 and 30 years old, which is also a little bit true for women butnot fully. For women the survival chances are higher between 14 and 40.

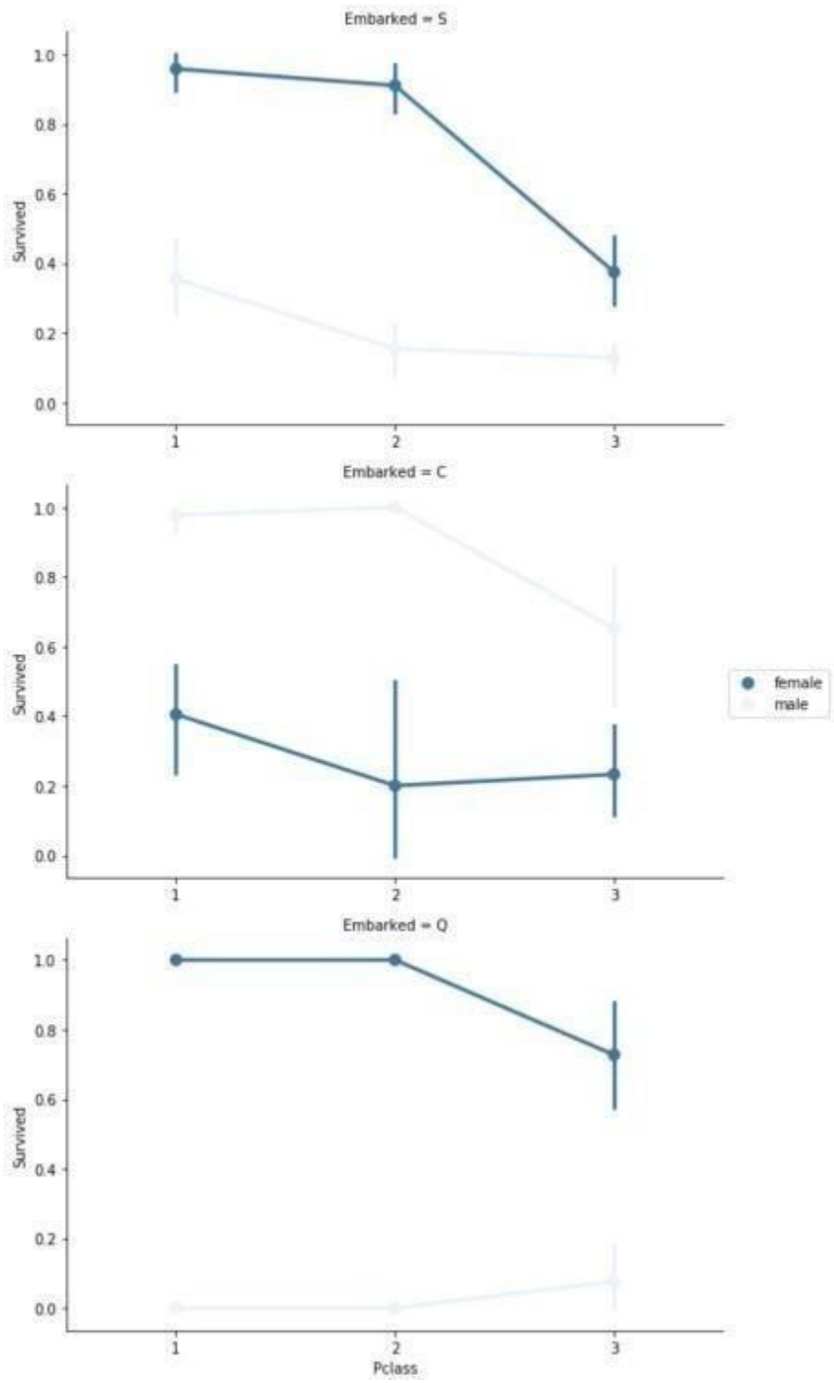
For men the probability of survival is very low between the age of 5 and 18,but that isn‟t true for women. Another thing to note is that infants also have a little bit higher probability of survival.

Since there seem to be **certain ages, which have increased odds ofsurvival** and because I want every feature to be roughly on the same scale, I will create age groups later on.

3. Embarked, Pclass and Sex:

```
FacetGrid = sns.FacetGrid(train_df, row='Embarked', size=4.5,
aspect=1.6)
FacetGrid.map(sns.pointplot, 'Pclass', 'Survived', 'Sex',
palette=None, order=None, hue_order=None ) FacetGrid.add_legend()
```

```
<seaborn.axisgrid.FacetGrid at 0x10ba485c0>
```



Embarked seems to be correlated with survival, depending on the gender.

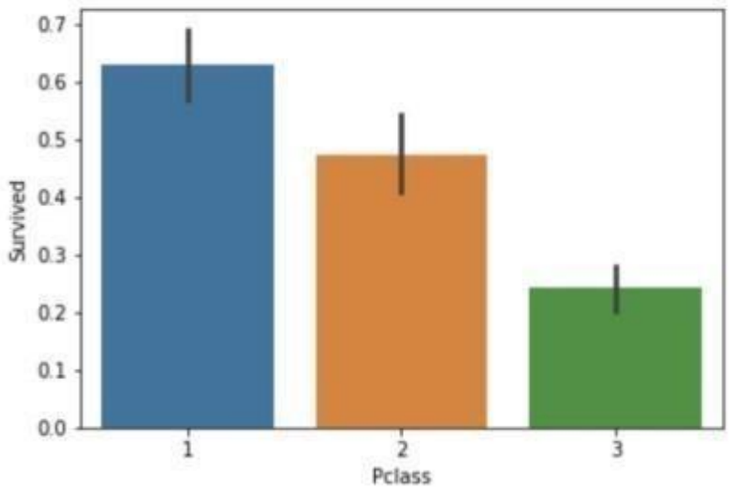
Women on port Q and on port S have a higher chance of survival. The inverse is true, if they are at port C. Men have a high survival probability ifthey are on port C, but a low probability if they are on port Q or S.

Pclass also seems to be correlated with survival. We will generate anotherplot of it below.

4. Pclass:

```
sns.barplot(x='Pclass',
                        y='Survived',data=train_df)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x10d1dc7b8>
```



Here we see clearly, that Pclass is contributing to a persons chance of survival, especially if this person is in class 1. We will create another pclassplot below.