

New chat



```
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import io
df = pd.read_csv('/content/uber.csv')
df.shape
df.head()
df.info()
df.isnull().sum() #Find if there are any null values present

#Drop null rows
df.dropna(axis=0,inplace=True)
df.isnull().sum()
#Calculate the distance between the pickup and dropoff co-ordinates
#Using the Haversine formula for accuracy.
def haversine (lon_1, lon_2, lat_1, lat_2):

    lon_1, lon_2, lat_1, lat_2 = map(np.radians, [lon_1, lon_2, lat_1, lat_2]) #Degrees to Radians

    diff_lon = lon_2 - lon_1
    diff_lat = lat_2 - lat_1

    km = 2 * 6371 * np.arcsin(np.sqrt(np.sin(diff_lat/2.0)**2 +
                                     np.cos(lat_1) * np.cos(lat_2) * np.sin(diff_lon/2.0)**2))

    return km
#find distance travelled per ride
df['Distance']= haversine(df['pickup_longitude'],df['dropoff_longitude'],
                          df['pickup_latitude'],df['dropoff_latitude'])
#round it to 2 decimal points
df['Distance'] = df['Distance'].astype(float).round(2)
df.head()
plt.scatter(df['Distance'], df['fare_amount'])
plt.xlabel("Distance")
plt.ylabel("fare_amount")
#Outliers
#We can get rid of the trips with very large distances that are outliers
# As well as trips with 0 distance.
df.drop(df[df['Distance'] > 60].index, inplace = True)
df.drop(df[df['Distance'] == 0].index, inplace = True)
```

```

df.drop(df[df['fare_amount'] == 0].index, inplace = True)
df.drop(df[df['fare_amount'] < 0].index, inplace = True)
df.shape
# Removing rows with non-plausible fare amounts and distance travelled
df.drop(df[(df['fare_amount'] > 100) & (df['Distance'] < 1)].index, inplace = True )
df.drop(df[(df['fare_amount'] < 100) & (df['Distance'] > 100)].index, inplace = True )
df.shape
plt.scatter(df['Distance'], df['fare_amount'])
plt.xlabel("Distance")
plt.ylabel("fare_amount")

df.info()
# Create New DataFrame of Specific column
df2 = pd.DataFrame().assign(fare=df['fare_amount'], Distance=df['Distance'])
df2.info()

df2.shape
# plot target fare distribution
plt.figure(figsize=[8,4])
sns.distplot(df2['fare'], color='g', hist_kws=dict(edgecolor="black", linewidth=2), bins=30)
plt.title('Target Variable Distribution')
plt.show()
x=df2['fare'] # Independent Variable
y=df2['Distance'] # Dependent Variable
#Independent variable
X = df2['Distance'].values.reshape(-1, 1)
#Dependent Variable
Y= df2['fare'].values.reshape(-1, 1)
# scale by StandardScaler
from sklearn.preprocessing import StandardScaler
std = StandardScaler()
y_std = std.fit_transform(Y)
x_std = std.fit_transform(X)
#Split into Training & Testing
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(x_std, y_std, test_size=0.2, random_state=0)
#simple linear regression
from sklearn.linear_model import LinearRegression
l_reg = LinearRegression()
l_reg.fit(X_train, y_train)
#predict test values
y_pred = l_reg.predict(X_test)
#Find the error btwn actual values and predicted values of the test
from sklearn import metrics
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
#final plot
plt.subplot(2, 2, 1)
plt.scatter(X_train, y_train, color = 'red')
plt.plot(X_train, l_reg.predict(X_train), color ="blue")

```

```
plt.title("Fare vs Distance (Training Set)")
plt.ylabel("fare_amount")
plt.xlabel("Distance")

plt.subplot(2, 2, 2)
plt.scatter(X_test, y_test, color = 'red')
plt.plot(X_train, l_reg.predict(X_train), color = "blue")
plt.ylabel("fare_amount")
plt.xlabel("Distance")
plt.title("Fare vs Distance (Test Set)")
```



Your code is set up well to analyze the relationship between the distance traveled and the fare amount in an Uber dataset using linear regression. Here are some key points that will improve your code and make it clearer:

Code Enhancements

1. **Check for file path issues:** Make sure the file path (``/content/uber.csv``) is correct if this is not running in a Google Colab or similar environment.
2. **Use `sns.histplot`** instead of `sns.distplot`: `distplot` is deprecated in newer versions of Seaborn. Replace it with:


```
python

sns.histplot(df2['fare'], color='g', edgecolor="black", bins=30)
```
3. **Set the Random State for Consistent Results:** You've used `random_state=0` for reproducibility in `train_test_split`, which is good.
4. **Add Comments to Explain Code Blocks:** Consider breaking down the code into sections with comments that clearly indicate the purpose of each step, e.g., "Data Cleaning," "Feature Engineering," etc.
5. **Import StandardScaler and Linear Regression at the Top:** This would make the code more organized and



expalin line by line



Let's go through each line step-by-step to understand what the code does:

Imports and Initial Setup

python

```
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

- ``numpy`` and ``pandas`` are imported for numerical and data manipulation tasks.
- ``math`` library is used for mathematical operations (though it's unused in this code).
- ``matplotlib.pyplot`` and ``seaborn`` are imported for data visualization.
- ``%matplotlib inline`` allows plots to display inline in Jupyter notebooks.

Loading and Initial Exploration of the Data

python

```
df = pd.read_csv('/content/uber.csv')
df.shape
df.head()
df.info()
df.isnull().sum() # Find if there are any null values present
```

- ``pd.read_csv`` reads in the CSV file and stores it in the dataframe ``df``.
- ``df.shape`` shows the dimensions (rows and columns) of the dataset.
- ``df.head()`` displays the first few rows to get a sense of the data structure.
- ``df.info()`` provides an overview of column data types and non-null counts.
- ``df.isnull().sum()`` counts missing values in each column to check for any null values.

Data Cleaning: Dropping Rows with Null Values

python

```
df.dropna(axis=0, inplace=True)
df.isnull().sum()
```

- ``df.dropna(axis=0, inplace=True)`` removes rows with any null values, modifying ``df`` in place.
- ``df.isnull().sum()`` verifies that there are no null values remaining.

Calculating Distance Between Coordinates Using the Haversine Formula

python

```
def haversine(lon_1, lon_2, lat_1, lat_2):
    lon_1, lon_2, lat_1, lat_2 = map(np.radians, [lon_1, lon_2, lat_1, lat_2]) # Degrees
    to Radians
    diff_lon = lon_2 - lon_1
    diff_lat = lat_2 - lat_1
```

```
km = 2 * 6371 * np.arcsin(np.sqrt(np.sin(diff_lat/2.0)**2 +
                                np.cos(lat_1) * np.cos(lat_2) * np.sin(diff_lon/2.0)
                                **2))
return km
```

- The `haversine` function calculates the distance between two geographic coordinates (longitude and latitude) using the Haversine formula.
- Coordinates are converted from degrees to radians.
- `diff_lon` and `diff_lat` store the differences between the pickup and dropoff points.
- The distance `km` is calculated by applying the Haversine formula, which uses the Earth's radius (6371 km).

Applying the Haversine Formula and Rounding Distances

python

```
df['Distance'] = haversine(df['pickup_longitude'], df['dropoff_longitude'],
                           df['pickup_latitude'], df['dropoff_latitude'])
df['Distance'] = df['Distance'].astype(float).round(2)
df.head()
```

- `df['Distance']` applies the `haversine` function to calculate the distance between pickup and dropoff points for each ride.
- `.astype(float).round(2)` ensures the `Distance` column is in float format, rounded to 2 decimal places.
- `df.head()` displays the updated dataframe with the `Distance` column.

Visualizing Distance vs Fare Amount

python

```
plt.scatter(df['Distance'], df['fare_amount'])
plt.xlabel("Distance")
plt.ylabel("fare_amount")
```

- This scatter plot visualizes the relationship between `Distance` and `fare_amount`.

Removing Outliers in Distance and Fare

python

```
df.drop(df[df['Distance'] > 60].index, inplace=True)
df.drop(df[df['Distance'] == 0].index, inplace=True)
df.drop(df[df['fare_amount'] == 0].index, inplace=True)
df.drop(df[df['fare_amount'] < 0].index, inplace=True)
df.shape
```

- Rows with distances greater than 60 km or less than 1 km and fares less than or equal to zero are removed as outliers.
- `df.shape` checks the new dimensions after filtering.

Removing Implausible Combinations of Distance and Fare

python

```
df.drop(df[(df['fare_amount'] > 100) & (df['Distance'] < 1)].index, inplace=True)
df.drop(df[(df['fare_amount'] < 100) & (df['Distance'] > 100)].index, inplace=True)
df.shape
```

- Removes rows where fare is unusually high for short distances or low for long distances.

Plotting Cleaned Data

python

```
plt.scatter(df['Distance'], df['fare_amount'])
plt.xlabel("Distance")
plt.ylabel("fare_amount")
df.info()
```

- Creates a scatter plot of `Distance` vs `fare_amount` for the cleaned dataset.

Creating a New DataFrame for Fare and Distance

python

```
df2 = pd.DataFrame().assign(fare=df['fare_amount'], Distance=df['Distance'])
df2.info()
df2.shape
```

- A new dataframe `df2` is created with only the `fare` and `Distance` columns for simplicity.

Visualizing Fare Distribution

python

```
plt.figure(figsize=[8,4])
sns.distplot(df2['fare'], color='g', hist_kws=dict(edgecolor="black", linewidth=2), bins=30)
plt.title('Target Variable Distribution')
plt.show()
```

- The distribution of fare amounts is plotted to observe data spread.

Defining Independent and Dependent Variables

python

```
x = df2['fare'] # Independent Variable
y = df2['Distance'] # Dependent Variable
```

```
X = df2['Distance'].values.reshape(-1, 1)
Y = df2['fare'].values.reshape(-1, 1)
```

- `X` and `Y` are reshaped as 2D arrays for compatibility with scikit-learn.

Scaling Features Using StandardScaler

python

```
from sklearn.preprocessing import StandardScaler
std = StandardScaler()
y_std = std.fit_transform(Y)
x_std = std.fit_transform(X)
```

- `StandardScaler` scales both `X` and `Y` for standardization, improving regression accuracy.

Splitting Data into Training and Testing Sets

python

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(x_std, y_std, test_size=0.2, random_state=0)
```

- `train_test_split` divides the data into training (80%) and testing (20%) sets for model validation.

Building and Training the Linear Regression Model

python

```
from sklearn.linear_model import LinearRegression
l_reg = LinearRegression()
l_reg.fit(X_train, y_train)
```

- Initializes and trains a simple linear regression model on the training data.

Making Predictions and Evaluating Model Performance

python

```
y_pred = l_reg.predict(X_test)
from sklearn import metrics
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
```

- `y_pred` stores the model's predictions on `X_test`.

- Mean Absolute Error, Mean Squared Error, and Root Mean Squared Error evaluate the model's accuracy.

Plotting the Training and Test Set Results

python

```
plt.subplot(2, 2, 1)
plt.scatter(X_train, y_train, color='red')
plt.plot(X_train, l_reg.predict(X_train), color="blue")
plt.title("Fare vs Distance (Training Set)")
plt.ylabel("fare_amount")
plt.xlabel("Distance")

plt.subplot(2, 2, 2)
plt.scatter(X_test, y_test, color='red')
plt.plot(X_train, l_reg.predict(X_train), color="blue")
plt.ylabel("fare_amount")
plt.xlabel("Distance")
plt.title("Fare vs Distance (Test Set)")
```

- Plots the regression line on both the training and test sets to visualize how well the model fits the data.

This completes a full analysis, including data preprocessing, feature engineering, model training, and evaluation. Let me know if you have any questions on specific steps!



```
import os
import string
from nltk.corpus import stopwords
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
df = pd.read_csv('/content/spam_ham_dataset.csv')
df.head()
df = df.drop(['Unnamed: 0'], axis=1)
df.head()
print('Total %s data email'% len(df))
#Total class members
df['label'].value_counts()
#show graph
df_label = sns.countplot(df['label'])
df_label.set_xticklabels(df['label'].unique())
plt.show()
#data preprocessing
#data text cleaning
# punctuations
punct = []
```



```

for char in string.punctuation:
    punct.append(char)
import re
def cleaning(txt):
    # case folding
    text = txt.lower()

    # remove multiple space, tabs, dan newlines
    text = re.sub('\s+', ' ', text)
    # remove links
    text = text.replace("http://", " ").replace("https://", " ")
    # remove special characters
    text = text.encode('ascii', 'replace').decode('ascii')
    text = ' '.join(re.sub("([@#][A-Za-z0-9]+)|(\w+:\V\S+)", " ", text).split())
    # remove punctuation
    text = ".join([word for word in text if word not in punct])
    #remove single character
    text = re.sub(r"\b[a-zA-Z]\b", "", text)
    #remove numbers
    text = re.sub(r"\d+", "", text)
    #remove multiple spaces (again)
    text = re.sub('\s+', ' ', text)
    return text
# call function for cleaning
# apply fungsi cleaning ke setiap text
df['text_cleaned'] = df['text'].apply(lambda x: cleaning(x))
df = df[['text', 'text_cleaned', 'label']]
df.head()
#compare
print(df['text'][0])
print(df['text_cleaned'][0])
# to remove stop words
import nltk
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
from nltk.corpus import stopwords
stop = stopwords.words('english')
df['text_cleaned'] = df['text_cleaned'].apply(lambda x: ' '.join([word for word in x.split() if word not
in stop]))
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')
from nltk.corpus import wordnet
nltk.download('omw-1.4')
#Lemmatization
from nltk.stem import WordNetLemmatizer # Import the WordNetLemmatizer class
lemmatizer = WordNetLemmatizer()
def get_wordnet_pos(word):
    """Map POS tag to first character lemmatize() accepts"""
    tag = nltk.pos_tag([word])[0][1][0].upper()

```

```

tag_dict = {"J": wordnet.ADJ,
            "N": wordnet.NOUN,
            "V": wordnet.VERB,
            "R": wordnet.ADV}
return tag_dict.get(tag, wordnet.NOUN)

def do_lemma(string):
    lemmatized = ' '.join([lemmatizer.lemmatize(word, get_wordnet_pos(word)) for word in
nlk.word_tokenize(string)])
    return lemmatized
df['text_cleaned'] = df['text_cleaned'].apply(lambda x: do_lemma(x))
df = df.drop(['text'], axis=1)
df = df.rename(columns = {'text_cleaned' : 'text'})
df.columns
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer()
X = tfidf.fit_transform(df['text'])
y = df['label']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
from time import time
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
# defining the classifier
clf = KNeighborsClassifier(n_neighbors=5, metric='euclidean')
#predicting the time of train and testing
t0 = time()
clf.fit(X_train, y_train)
print("\nTraining time:", round(time()-t0, 3), "s\n")
#predicting the time of testing
t1 = time()
pred = clf.predict(X_test)
print("Predicting time:", round(time()-t1, 3), "s\n")
#calculating and printing the accuracy of the algorithm
print("Accuracy of KNN Algorithm: ", accuracy_score(pred,y_test)) explain line by line in simple
words

```



Let's go through each line and section of the code to understand what each part is doing, explained step-by-step in simple terms:

Imports

python

```

import os
import string
from nltk.corpus import stopwords
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import numpy as np

```

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

- This section imports various libraries:
 - ``os`, `string``: for basic operations and string handling.
 - ``nlk.corpus.stopwords``: for handling stopwords in text.
 - ``train_test_split`` and ``accuracy_score`` from ``sklearn`` for splitting data and measuring model performance.
 - ``numpy`, `pandas``: for data handling and manipulation.
 - ``seaborn`, `matplotlib.pyplot``: for data visualization.

Loading the Data

python

```
df = pd.read_csv('/content/spam_ham_dataset.csv')
df.head()
```

- Reads in the dataset from a CSV file and displays the first few rows using ``df.head()``.

Dropping Unnecessary Columns

python

```
df = df.drop(['Unnamed: 0'], axis=1)
df.head()
```

- Removes the ``Unnamed: 0`` column, which is likely an unnecessary index column.

Checking Dataset Size and Class Distribution

python

```
print('Total %s data email' % len(df))
df['label'].value_counts()
```

- Prints the total number of rows (emails) in the dataset.
- ``df['label'].value_counts()`` shows how many spam and ham (non-spam) emails are in the dataset.

Plotting Class Distribution

python

```
df_label = sns.countplot(df['label'])
df_label.set_xticklabels(df['label'].unique())
```

```
plt.show()
```

- Creates a bar plot showing the distribution of spam and ham emails in the dataset.

Defining Punctuation and Cleaning Text Function

python

```
punct = []
for char in string.punctuation:
    punct.append(char)
import re
```

- ``punct`` stores all punctuation characters to help with text cleaning.
- ``re`` is imported for regular expression handling in text cleaning.

Cleaning Text Function

python

```
def cleaning(txt):
    text = txt.lower() # Convert to lowercase
    text = re.sub('\s+', ' ', text) # Remove extra spaces, tabs, newlines
    text = text.replace("http://", " ").replace("https://", " ") # Remove links
    text = text.encode('ascii', 'replace').decode('ascii') # Remove non-ASCII characters
    text = ' '.join(re.sub("([@#][A-Za-z0-9+])|(\w+:\/\/\w+)", " ", text).split()) # Remove
    # Remove hashtags and mentions
    text = ''.join([word for word in text if word not in punct]) # Remove punctuation
    text = re.sub(r"\b[a-zA-Z]\b", "", text) # Remove single characters
    text = re.sub(r"\d+", "", text) # Remove numbers
    text = re.sub('\s+', ' ', text) # Remove extra spaces again
    return text
```

- The ``cleaning`` function prepares the text for analysis:
 - Converts text to lowercase.
 - Removes extra spaces, links, non-ASCII characters, hashtags, mentions, punctuation, single characters, and numbers.

Applying the Cleaning Function

python

```
df['text_cleaned'] = df['text'].apply(lambda x: cleaning(x))
df = df[['text', 'text_cleaned', 'label']]
df.head()
```

- Applies the ``cleaning`` function to each email in the ``text`` column and stores the cleaned text in a new column ``text_cleaned``.
- Keeps only the ``text``, ``text_cleaned``, and ``label`` columns.

Comparing Original and Cleaned Text

python

```
print(df['text'][0])
print(df['text_cleaned'][0])
```

- Prints the original and cleaned versions of the first email to see the effect of cleaning.

Removing Stopwords

python

```
import nltk
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
from nltk.corpus import stopwords
stop = stopwords.words('english')
df['text_cleaned'] = df['text_cleaned'].apply(lambda x: ' '.join([word for word in x.split() if word not in stop]))
```

- Downloads necessary NLTK resources.
- Removes common "stop words" (e.g., "and", "the") that don't add much meaning to the text by applying a filter on the `text_cleaned` column.

Lemmatization for Root Words

python

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()

def get_wordnet_pos(word):
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ, "N": wordnet.NOUN, "V": wordnet.VERB, "R": wordnet.ADV}
    return tag_dict.get(tag, wordnet.NOUN)

def do_lemma(string):
    lemmatized = ' '.join([lemmatizer.lemmatize(word, get_wordnet_pos(word)) for word in nltk.word_tokenize(string)])
    return lemmatized

df['text_cleaned'] = df['text_cleaned'].apply(lambda x: do_lemma(x))
```

- `do_lemma` function lemmatizes each word in `text_cleaned` by reducing it to its root form (e.g., "running" becomes "run").
- Applies `do_lemma` to each email.

Final Column Cleanup

python

```
df = df.drop(['text'], axis=1)
df = df.rename(columns={'text_cleaned': 'text'})
```

```
df.columns
```

- Removes the original `text` column, keeps only `text_cleaned` (now renamed as `text`), and `label`.

Converting Text to Numerical Features with TF-IDF

```
python
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer()
X = tfidf.fit_transform(df['text'])
y = df['label']
```

- Uses `TfidfVectorizer` to convert text into numerical data for model training, where `X` is the matrix of TF-IDF features and `y` contains labels (spam or ham).

Splitting Data into Training and Testing Sets

```
python
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

- Splits data into training (80%) and testing (20%) sets.

K-Nearest Neighbors Model Training and Prediction

```
python
```

```
from time import time
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

clf = KNeighborsClassifier(n_neighbors=5, metric='euclidean') # Defines the KNN classifier with 5 neighbors

t0 = time()
clf.fit(X_train, y_train) # Trains the classifier
print("\nTraining time:", round(time() - t0, 3), "s\n")

t1 = time()
pred = clf.predict(X_test) # Makes predictions on the test set
print("Predicting time:", round(time() - t1, 3), "s\n")
```

- Initializes a K-Nearest Neighbors (KNN) classifier with 5 neighbors.
- Trains the classifier on the training set and measures the training time.
- Makes predictions on the test set and measures the prediction time.

Evaluating Model Accuracy

```
python
```

```
print("Accuracy of KNN Algorithm: ", accuracy_score(pred, y_test))
```

- `accuracy_score` calculates the model's accuracy by comparing predictions (`pred`) to actual labels (`y_test`).

This completes the end-to-end process, including data loading, cleaning, feature extraction, training, and model evaluation. Let me know if you need more clarification on any part!



```
# Calling all the dependencies for this project
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
df = pd.read_csv('/content/Churn_Modelling.csv')
df.shape
df.drop(['CustomerId','RowNumber','Surname'], axis = 'columns', inplace =True)
df.isna().sum()
df.dtypes
df['Geography'].unique()
df['Gender'].unique()
df['Gender'].replace(['Male', 'Female'],[1, 0], inplace= True)
#one hot encoding
df = pd.get_dummies(data = df, columns=['Geography'])
df.dtypes
df['Exited'].value_counts()
def visualization(x, y, xlabel):
    plt.figure(figsize=(10,5))
    plt.hist([x, y], color=['red', 'green'], label = ['exit', 'not_exit'])
    plt.xlabel(xlabel,fontsize=20)
    plt.ylabel("No. of customers", fontsize=20)
    plt.legend()
#get in to temp fframe
df_exited = df[df['Exited']==1]['Tenure']
df_not_exited = df[df['Exited']==0]['Tenure']
visualization(df_exited, df_not_exited, "Tenure")
df_exited1 = df[df['Exited']==1]['Age']
df_not_exited1 = df[df['Exited']==0]['Age']
visualization(df_exited1, df_not_exited1, "Tenure")
# normalization
cols_to_scale = ['CustomerId','CreditScore', 'Age','Tenure', 'Balance', 'NumOfProducts',
'EstimatedSalary']
from sklearn.preprocessing import MinMaxScaler
```

```
# Define the scaler
scaler = MinMaxScaler()

# Select only the numerical features for scaling
numerical_features = ['CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'EstimatedSalary']

# Fit the scaler on the numerical features and transform them
df[numerical_features] = scaler.fit_transform(df[numerical_features])
#separate outcome or target col
X = df.drop(['Exited'], axis=1)
y = df['Exited']
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_state=0)
from sklearn.preprocessing import StandardScaler
# feature scaling

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
import tensorflow as tf
from tensorflow import keras
model = keras.Sequential([
    keras.layers.Dense(12, input_shape=(12,), activation='relu'),
    keras.layers.Dense(15, activation='relu'),
    keras.layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.fit(X_train, y_train, epochs=100)
model.evaluate(X_test, y_test)
yp=model.predict(X_test)
from sklearn.metrics import confusion_matrix , classification_report
import numpy as np

# Convert predicted probabilities to class labels using a threshold (e.g., 0.5)
y_pred = np.where(yp > 0.5, 1, 0) # Assuming binary classification

# Now use yp_classes in the classification_report function
print(classification_report(y_test, y_pred))
tf.math.confusion_matrix(labels=y_test, predictions=y_pred) explain line by line in simple words
```



Here's an explanation of each step in this code, line by line:

1. Import Libraries:

python

```
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
import seaborn as sns
```

- Imports essential libraries for data analysis (`pandas`), numerical operations (`numpy`), and visualization (`matplotlib`, `seaborn`).

2. Load Dataset:

python

```
df = pd.read_csv('/content/Churn_Modelling.csv')
df.shape
```

- Reads the dataset into a DataFrame (`df`) and checks its dimensions.

3. Drop Irrelevant Columns:

python

```
df.drop(['CustomerId', 'RowNumber', 'Surname'], axis='columns', inplace=True)
```

- Removes columns that are not useful for the analysis (`CustomerId`, `RowNumber`, `Surname`).

4. Check for Missing Values:

python

```
df.isna().sum()
```

- Counts any missing values in each column to ensure data quality.

5. Check Data Types:

python

```
df.dtypes
```

- Lists the data types of each column to identify columns that may need to be transformed.

6. Check Unique Values in Categorical Columns:

python

```
df['Geography'].unique()
df['Gender'].unique()
```

- Prints unique values in `Geography` and `Gender` columns to understand categories.

7. Encode Gender Column:

python

```
df['Gender'].replace(['Male', 'Female'], [1, 0], inplace=True)
```

- Converts `Gender` column to numerical format, where `Male` is `1` and `Female` is `0`.

8. One-Hot Encoding for Geography:

python

```
df = pd.get_dummies(data=df, columns=['Geography'])
```

- Converts the `Geography` column into multiple binary columns (one for each unique value).

9. Check Class Distribution:

python

```
df['Exited'].value_counts()
```

- Counts how many customers exited or not, showing class balance for the target variable.

10. Define Visualization Function:

python

```
def visualization(x, y, xlabel):
    plt.figure(figsize=(10,5))
    plt.hist([x, y], color=['red', 'green'], label=['exit', 'not_exit'])
    plt.xlabel(xlabel, fontsize=20)
    plt.ylabel("No. of customers", fontsize=20)
    plt.legend()
```

- Defines a function to plot histograms for customers who exited vs. did not exit, given a feature.

11. Plot for Tenure and Age:

python

```
df_exited = df[df['Exited']==1]['Tenure']
df_not_exited = df[df['Exited']==0]['Tenure']
visualization(df_exited, df_not_exited, "Tenure")

df_exited1 = df[df['Exited']==1]['Age']
df_not_exited1 = df[df['Exited']==0]['Age']
visualization(df_exited1, df_not_exited1, "Tenure")
```

- Splits data based on `Exited` status and visualizes `Tenure` and `Age` distributions for both groups.

12. Normalize Features:

python

```
cols_to_scale = ['CustomerId', 'CreditScore', 'Age', 'Tenure', 'Balance', 'NumOfProduct
s', 'EstimatedSalary']
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
df[numerical_features] = scaler.fit_transform(df[numerical_features])
```

- Normalizes continuous columns with `MinMaxScaler` to bring values to a 0-1 range.

13. Separate Features and Target:

python

```
X = df.drop(['Exited'], axis=1)
y = df['Exited']
```

- Sets `X` as all features except `Exited`, and `y` as the `Exited` target variable.

14. Split Data into Train and Test Sets:

python

```
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
=0)
```

- Divides data into training (80%) and testing (20%) sets.

15. Feature Scaling for Neural Network:

python

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

```
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

- Standardizes the data to have a mean of 0 and standard deviation of 1, which helps with neural network training.

16. Define the Neural Network Model:

python

```
import tensorflow as tf
from tensorflow import keras
model = keras.Sequential([
    keras.layers.Dense(12, input_shape=(12,)), activation='relu'),
    keras.layers.Dense(15, activation='relu'),
    keras.layers.Dense(1, activation='sigmoid')
])
```

- Sets up a simple neural network model with 3 layers. The last layer has a sigmoid activation for binary classification.

17. Compile the Model:

python

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

- Compiles the model with `adam` optimizer and binary cross-entropy loss, which is common for binary classification.

18. Train the Model:

python

```
model.fit(X_train, y_train, epochs=100)
```

- Trains the model on `X_train` and `y_train` for 100 epochs.

19. Evaluate Model Performance:

python

```
model.evaluate(X_test, y_test)
```

- Tests the model on `X_test` and `y_test` to assess its performance, showing loss and accuracy.

20. Predict on Test Data:

python

```
yp = model.predict(X_test)
```

- Predicts churn probability for each test data point.

21. Convert Predictions to Binary Labels:

python

```
y_pred = np.where(yp > 0.5, 1, 0)
```

- Converts probabilities to binary class labels (1 or 0) using a threshold of 0.5.

22. Classification Report:

python

```
from sklearn.metrics import confusion_matrix, classification_report
print(classification_report(y_test, y_pred))
```

- Generates a report with precision, recall, F1-score, and support for each class.

23. Confusion Matrix:

python

```
tf.math.confusion_matrix(labels=y_test, predictions=y_pred)
```

- Prints a confusion matrix showing correct and incorrect predictions for each class.



```
cur_x = 2 # The algorithm starts at x=3
rate = 0.01 # Learning rate
precision = 0.000001 #This tells us when to stop the algorithm
previous_step_size = 1 #
max_iters = 10000 # maximum number of iterations
iters = 0 #iteration counter
df = lambda x: 2*(x+3) #Gradient of our function
while previous_step_size > precision and iters < max_iters:
    prev_x = cur_x #Store current x value in prev_x
    cur_x = cur_x - rate * df(prev_x) #Grad descent
    previous_step_size = abs(cur_x - prev_x) #Change in x
    iters = iters+1 #iteration count
```

```
print("Iteration",iters,"\nX value is",cur_x) #Print iterations
print("The local minimum occurs at", cur_x) explain line by line in simple words
```



Here's a breakdown of the code, line by line, to understand how gradient descent is being used to find a local minimum:

1. Initialize Variables:

```
python

cur_x = 2 # The algorithm starts at x=2
rate = 0.01 # Learning rate
precision = 0.000001 # Determines the stopping criteria for the algorithm
previous_step_size = 1 # Initial step size, set to a large value to enter the loop
max_iters = 10000 # Maximum iterations allowed to prevent infinite looping
iters = 0 # Counter to track the number of iterations
```

- These variables control the behavior of the gradient descent algorithm. `cur_x` is the starting point, `rate` sets the step size, and `precision` sets how small the step size must get for the loop to stop. `max_iters` prevents the loop from running indefinitely.

2. Define the Gradient Function:

```
python

df = lambda x: 2*(x+3) # Gradient of our function
```

- This is a lambda function representing the derivative of the function we are trying to minimize. It calculates the gradient, which points in the direction of the steepest increase, for a given `x`. The formula here, `2*(x+3)`, would be the derivative of the function $f(x) = (x+3)^2$.

3. Gradient Descent Loop:

```
python

while previous_step_size > precision and iters < max_iters:
```

- The loop runs until either the change in `x` (`previous_step_size`) is less than the desired `precision`, or the algorithm reaches the `max_iters` limit.

4. Store Current x:

python

```
prev_x = cur_x # Store current x value in prev_x
```

- Saves the current value of `cur_x` in `prev_x` before updating `cur_x`, to track the change in `x`.

5. Update x Using Gradient Descent Formula:

python

```
cur_x = cur_x - rate * df(prev_x) # Gradient descent step
```

- Updates `cur_x` by moving against the gradient. `rate * df(prev_x)` calculates the step size. Subtracting it from `cur_x` ensures we move in the direction of decrease (minimizing the function).

6. Calculate Step Size:

python

```
previous_step_size = abs(cur_x - prev_x) # Change in x
```

- Computes the change in `x` by finding the absolute difference between `cur_x` and `prev_x`. This value determines if the loop should stop if the change is small enough.

7. Increment Iteration Counter:

python

```
iters = iters + 1 # Increment iteration count
```

- Increases the iteration count by 1 each time the loop executes.

8. Print Current State:

python

```
print("Iteration", iters, "\nX value is", cur_x) # Print iteration info
```

- Prints the current iteration and the value of `cur_x` after each update.

9. Final Output:

```
python
```

```
print("The local minimum occurs at", cur_x)
```

- Once the loop ends, the program prints the value of `cur_x`, which is the estimated position of the local minimum for the function being minimized.



```
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
df = pd.read_csv('/content/diabetes.csv')
df.head()
df.shape
df.drop(['Pregnancies', 'BloodPressure', 'SkinThickness'], axis=1, inplace=True)
df.info()
df.describe()
#Aiming to impute nan values for the columns in accordance
#With their distribution
df[['Glucose', 'Insulin', 'BMI']].replace(0, np.NaN)
columns = ['Glucose', 'Insulin', 'BMI']
for col in columns:
    val = df[col].mean()
    df[col].replace(0, val)
#plot graph
graph = ['Glucose', 'Insulin', 'BMI', 'Age', 'Outcome']
sns.set()
print(sns.pairplot(df[graph], hue='Outcome', diag_kind='kde'))
#separate outcome or target col
X = df.drop(['Outcome'], axis=1)
y = df['Outcome']
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score
# Feature Scaling

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
classifier = KNeighborsClassifier(n_neighbors=11, p=2, metric='euclidean')
```



```

classifier.fit(X_train,y_train)
y_pred = classifier.predict(X_test)
# Evaluating the Model

conf_matrix = confusion_matrix(y_test,y_pred)
print(conf_matrix)
print(f1_score(y_test,y_pred))
# Accuracy
print(accuracy_score(y_test,y_pred))
# roc curve
from sklearn.metrics import roc_curve
plt.figure(dpi=100)
fpr, tpr, thresholds = roc_curve(y_test, y_pred)
from sklearn.metrics import roc_auc_score

temp=roc_auc_score(y_test,y_pred)

plt.plot(fpr,tpr,label = "%.2f" %temp)
plt.legend(loc = 'lower right')
plt.grid(True) explain line by line in simple words

```



Let's break down this code step-by-step to understand its purpose and functionality:

Import Libraries and Data

1. Library Imports:

```

python

import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

```

- Import necessary libraries for data manipulation, math functions, and visualization.

2. Load Dataset:

```

python

df = pd.read_csv('/content/diabetes.csv')
df.head()
df.shape

```

- Load the dataset named ``diabetes.csv`` and show the first few rows with ``df.head()``.
``df.shape`` shows the dataset's dimensions (rows, columns).

3. Remove Certain Columns:

python

```
df.drop(['Pregnancies', 'BloodPressure', 'SkinThickness'], axis=1, inplace=True)
df.info()
df.describe()
```

- Drop the columns `'Pregnancies'`, `'BloodPressure'`, and `'SkinThickness'` as they won't be used in this analysis. `df.info()` displays the data types, and `df.describe()` provides summary statistics.

Handle Missing Values

4. Replace Zeros with NaN for Specific Columns:

python

```
df[['Glucose', 'Insulin', 'BMI']].replace(0, np.NaN)
```

- Replace 0 values with `NaN` for the columns `'Glucose'`, `'Insulin'`, and `'BMI'` (assuming 0 is an invalid or missing value in these columns).

5. Impute Missing Values with Mean:

python

```
columns = ['Glucose', 'Insulin', 'BMI']
for col in columns:
    val = df[col].mean()
    df[col].replace(0, val)
```

- Loop over the columns `'Glucose'`, `'Insulin'`, and `'BMI'`, replacing any remaining 0 values in these columns with the column's mean to fill missing values.

Visualization

6. Plot Pair Plot:

python

```
graph = ['Glucose', 'Insulin', 'BMI', 'Age', 'Outcome']
sns.set()
print(sns.pairplot(df[graph], hue='Outcome', diag_kind='kde'))
```

- Define the list of columns `graph` to include in the pair plot. Use `sns.pairplot` to create a matrix of scatter plots between these variables, with separate colors for each `'Outcome'` (0 or 1). `diag_kind='kde'` specifies kernel density estimate plots on the diagonal.

Prepare Data for Model Training

7. Separate Features and Target:

python

```
X = df.drop(['Outcome'], axis=1)
y = df['Outcome']
```

- Separate the dataset into features `X` (all columns except `Outcome`) and the target `y` (the `Outcome` column).

8. Train-Test Split:

python

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

- Split the data into training and test sets, with 20% of the data reserved for testing. `random_state=0` ensures the split is reproducible.

9. Standardize Features:

python

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

- Apply feature scaling using `StandardScaler` to normalize the data. This helps the K-Nearest Neighbors (KNN) algorithm, which is sensitive to feature scales.

Model Training and Evaluation

10. K-Nearest Neighbors Classifier:

python

```
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors=11, p=2, metric='euclidean')
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
```

- Initialize the KNN classifier with 11 neighbors, using Euclidean distance (`p=2`) as the distance metric. Train (`fit`) the classifier on the training data and make predictions (`predict`) on the test data.

11. Confusion Matrix and F1 Score:

python

```
from sklearn.metrics import confusion_matrix, f1_score, accuracy_score
conf_matrix = confusion_matrix(y_test, y_pred)
print(conf_matrix)
print(f1_score(y_test, y_pred))
```

- Compute and display the confusion matrix, which shows counts of true positives, true negatives, false positives, and false negatives. Also, calculate and print the F1 score, a measure of accuracy considering both precision and recall.

12. Accuracy Score:

python

```
print(accuracy_score(y_test, y_pred))
```

- Calculate and print the accuracy, the ratio of correctly predicted instances out of the total instances.

ROC Curve and AUC

13. Plot ROC Curve:

python

```
from sklearn.metrics import roc_curve
plt.figure(dpi=100)
fpr, tpr, thresholds = roc_curve(y_test, y_pred)
```

- Import `roc_curve` to calculate the false positive rate (`fpr`), true positive rate (`tpr`), and thresholds. These values will be used to plot the ROC curve, which shows the trade-off between true positive rate and false positive rate.

14. Calculate AUC and Plot:

python

```
from sklearn.metrics import roc_auc_score
temp = roc_auc_score(y_test, y_pred)
plt.plot(fpr, tpr, label="%.2f" % temp)
plt.legend(loc='lower right')
plt.grid(True)
```

- Calculate the Area Under the Curve (AUC) score with `roc_auc_score`, a summary metric for the ROC curve where a higher score indicates better classification performance. Plot the ROC curve and label it with the AUC score.



continue



Certainly! Let's continue explaining the last part of the code:

ROC Curve and AUC (continued)

- **ROC Curve Explanation:**

python

```
plt.plot(fpr, tpr, label="%.2f" % temp)
plt.legend(loc='lower right')
plt.grid(True)
```

- **Plotting the ROC Curve:** The `plt.plot(fpr, tpr)` command plots the false positive rate (`fpr`) against the true positive rate (`tpr`) obtained earlier from the `roc_curve` function. The ROC curve shows how the classifier performs across different thresholds.
- **Labeling the AUC Score:** The `label="%.2f" % temp` part labels the curve with the AUC value (`temp`) formatted to two decimal places. This label helps you quickly identify the performance of the classifier.
- **Adding a Legend:** `plt.legend(loc='lower right')` places the legend in the lower right corner of the plot, showing the AUC score.
- **Adding a Grid:** `plt.grid(True)` adds a grid to the plot, making it easier to visualize and interpret the curve.

Final Outcome:

- **Understanding the Output:**

- **Confusion Matrix:** Shows the number of correct and incorrect predictions categorized by their type (True Positive, False Positive, True Negative, False Negative). This helps in evaluating how well the classifier performs.
- **F1 Score:** The harmonic mean of Precision and Recall, which provides a balance between the two. It's especially useful when the class distribution is imbalanced.
- **Accuracy Score:** The percentage of correct predictions out of all predictions. However, in imbalanced datasets, accuracy might not be a sufficient evaluation metric.
- **ROC Curve:** Plots the trade-off between the true positive rate and the false positive rate, helping you understand how well the model differentiates between the positive and negative classes.
- **AUC (Area Under the Curve):** The AUC score is a single value that summarizes the performance of the classifier across all possible classification thresholds. The higher the AUC, the better the model's ability to distinguish between the classes.

Recap of the Steps in the Code:

- 1. **Data Preprocessing:**

- Dropped unnecessary columns.

- Handled missing values by replacing zeros with the mean of respective columns.
- Visualized the relationships between features and target using pair plots.

2. Model Preparation:

- Separated features (`X`) and target (`y`).
- Split the dataset into training and testing sets.
- Standardized the features to ensure the KNN algorithm works well with scaled data.

3. Model Training and Evaluation:

- Trained a KNN classifier with `n_neighbors=11` and evaluated its performance using:
 - Confusion Matrix
 - F1 Score
 - Accuracy Score
 - ROC Curve and AUC

By following this process, you can assess how well your classifier performs and use these metrics to fine-tune the model for better prediction accuracy.



```
import numpy as np
import pandas as pd
import io
df = pd.read_csv('/content/Mall_Customers.csv')
df.shape
df.head()
df["A"] = df[["Annual Income (k$)"]]
df["B"] = df[["Spending Score (1-100)"]]
X = df[["A", "B"]]
X.head()
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
%matplotlib inline
plt.scatter(X["A"], X["B"], s = 30, c = 'b')
plt.show()
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler() # Removed the extra indent at the beginning of this line.
X_scaled = scaler.fit_transform(X)

Kmean = KMeans(n_clusters=5)
Kmean.fit(X)
centers = Kmean.cluster_centers_
print(Kmean.cluster_centers_)
clusters = Kmean.fit_predict(X)
df["label"] = clusters
df.head(100)
col = ['green', 'blue', 'black', 'yellow', 'orange']
for i in range(5):
    a = col[i]
    # print(a)
```

```
plt.scatter(df.A[df.label==i], df.B[df.label == i], c=a, label='cluster 1')
plt.scatter(centers[:, 0], centers[:, 1], marker='*', s=300,
            c='r', label='centroid')
X1 = X.loc[:,["A","B"]].values
wcss=[]
for k in range(1,11):
    kmeans = KMeans(n_clusters = k, init = "k-means++")
    kmeans.fit(X1)
    wcss.append(kmeans.inertia_)
plt.figure(figsize =( 12,6))
plt.grid()
plt.plot(range(1,11),wcss,linewidth=2,color="red",marker="8")
plt.xlabel("K Value")
plt.ylabel("WCSS")
plt.show()
```



Let's go through the code step by step and explain each part in simple words:

1. Loading the Data:

python

```
df = pd.read_csv('/content/Mall_Customers.csv')
df.shape
df.head()
```

- The `pd.read_csv('/content/Mall_Customers.csv')` line loads the dataset from a CSV file into a pandas DataFrame (`df`).
- `df.shape` returns the dimensions of the DataFrame (number of rows and columns).
- `df.head()` displays the first 5 rows of the dataset to get a quick look at the data.

2. Selecting the Columns:

python

```
df["A"] = df[["Annual Income (k$)"]]
df["B"] = df[["Spending Score (1-100)"]]
X = df[["A", "B"]]
X.head()
```

- Here, columns `'Annual Income (k$)'` and `'Spending Score (1-100)'` are selected and renamed to `'A'` and `'B'`, respectively.
- These columns are used for clustering.
- `X` is created as a subset of `df` containing only columns `'A'` and `'B'`, which are the features for clustering.
- `X.head()` displays the first few rows of this new DataFrame `X`.

3. Plotting the Data:

python

```
plt.scatter(X["A"], X["B"], s = 30, c = 'b')
plt.show()
```

- This creates a scatter plot of the data, where the x-axis represents "Annual Income (k\$)" and the y-axis represents "Spending Score (1-100)".
- The `s=30` sets the size of the points in the plot, and `c='b'` sets the color to blue.

4. Standardizing the Data:

python

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

- Standardization is a common preprocessing step where the features (data) are scaled to have a mean of 0 and a standard deviation of 1.
- `scaler.fit_transform(X)` scales the `X` data using the `StandardScaler` and stores the scaled version in `X_scaled`.

5. K-Means Clustering:

python

```
Kmean = KMeans(n_clusters=5)
Kmean.fit(X)
centers = Kmean.cluster_centers_
print(Kmean.cluster_centers_)
clusters = Kmean.fit_predict(X)
df["label"] = clusters
df.head(100)
```

- `KMeans(n_clusters=5)` initializes the KMeans algorithm with 5 clusters.
- `Kmean.fit(X)` fits the KMeans model to the data `X` to find clusters.
- `Kmean.cluster_centers_` stores the coordinates of the centroids (center points) of each of the 5 clusters.
- `Kmean.fit_predict(X)` computes the cluster labels for each data point (which cluster they belong to) and stores it in the variable `clusters`.
- The column `label` is added to the DataFrame `df` to store the cluster labels.
- `df.head(100)` shows the first 100 rows of the DataFrame with the added cluster labels.

6. Visualizing the Clusters:

python

```
col=['green','blue','black','yellow','orange',]
for i in range(5):
    a=col[i]
    plt.scatter(df.A[df.label==i], df.B[df.label == i], c=a, label=f'cluster {i+1}')
plt.scatter(centers[:, 0], centers[:, 1], marker='*', s=300, c='r', label='centroid')
```

- This loops through each of the 5 clusters (0 to 4) and creates a scatter plot of the points in each cluster using different colors (`col` list).
- The points belonging to each cluster are plotted in their respective colors.
- The `plt.scatter(centers[:, 0], centers[:, 1], marker='*', s=300, c='r', label='centroid')` command plots the centroids (the center points) of the clusters as red stars (`marker='*'`) with size 300.
- The `label=f'cluster {i+1}'` adds a label for each cluster (1 to 5).

7. Finding the Optimal Number of Clusters (Elbow Method):

python

```
X1 = X.loc[:,["A","B"]].values
wcss = []
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k, init="k-means++")
    kmeans.fit(X1)
    wcss.append(kmeans.inertia_)
plt.figure(figsize=(12,6))
plt.grid()
plt.plot(range(1, 11), wcss, linewidth=2, color="red", marker="8")
plt.xlabel("K Value")
plt.ylabel("WCSS")
plt.show()
```

- The `Elbow Method` is used to determine the optimal number of clusters. It does so by plotting the "Within-Cluster Sum of Squares" (WCSS) for different values of `k` (number of clusters).
- The `kmeans.inertia_` provides the WCSS for each value of `k`, which is the sum of squared distances of samples to their cluster center.
- The code runs KMeans clustering for `k=1` to `k=10` and stores the WCSS values in the list `wcss`.
- The plot (`plt.plot(range(1, 11), wcss)`) shows how WCSS changes as `k` increases.
- The "elbow" in the graph helps identify the ideal value of `k`, which is typically where the decrease in WCSS slows down.

Final Outcome:

- The **KMeans clustering** groups customers into 5 clusters based on their annual income and spending score.
- The **visualization** shows how the customers are distributed across the clusters and highlights the cluster centroids.
- The **Elbow Method** helps determine the optimal number of clusters by observing where the WCSS starts to level off.

This process can be applied to segment customers for marketing, identifying different consumer groups based on income and spending behavior.

ChatGPT can make mistakes. Check important info.