Name : A Manikandan

Batch : DA/DS November – 2024

Date : 06/06/2025

**Objective:**

* To identifying an e-commerce website that allows for web scraping and to extract essential product data
* To use necessary libraries and develop a script to scrape this data and save it in a CSV file
* To pre-process the data and perform EDA to understand it in-depth
* To Create and Evaluate 1 Unsupervised and 5 Supervised Classification model based on the data.

**Libraries used:**

from selenium import webdriver

import pandas as pd

import numpy as np

from bs4 import BeautifulSoup

import time

from webdriver\_manager.chrome import ChromeDriverManager

from selenium.webdriver.chrome.service import Service

from selenium.webdriver.chrome.options import Options

from matplotlib import pyplot as plt

import seaborn as sns

from sklearn.datasets import make\_blobs

from sklearn.cluster import KMeans

from pyclustertend import hopkins

from sklearn.preprocessing import LabelEncoder, MinMaxScaler

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.neighbors import KNeighborsClassifier as KNN

from sklearn.ensemble import RandomForestClassifier

import xgboost as xgb

from sklearn.model\_selection import GridSearchCV, RandomizedSearchCV, cross\_val\_score, KFold, train\_test\_split

from sklearn.metrics import mean\_squared\_error, r2\_score, f1\_score,accuracy\_score, classification\_report, confusion\_matrix,silhouette\_score

import warnings

warnings.filterwarnings("ignore")

**Website:**

The E-commerce website used for scrapping is Flipkart and the product chosen is wrist watch.

The URL’s hyperlink used is provided below for reference – [Flipkart Wrist Watch](https://www.flipkart.com/search?q=watches+for+women&sid=r18%2Cf13&as=on&as-show=on&otracker=AS_QueryStore_OrganicAutoSuggest_2_7_na_na_ps&otracker1=AS_QueryStore_OrganicAutoSuggest_2_7_na_na_ps&as-pos=2&as-type=RECENT&suggestionId=watches+for+women%7CWrist+Watches&requestId=459df78d-0a5e-48f0-97b3-58d0b514e43c&as-searchtext=watches&p%5B%5D=facets.ideal_for%255B%255D%3DCouple&p%5B%5D=facets.ideal_for%255B%255D%3DWomen&p%5B%5D=facets.ideal_for%255B%255D%3DMen)

options = Options()

options.add\_argument("--headless")

options.add\_argument("--disable-gpu")

This code is a Selenium-based web scraper setup that launches a headless Chrome browser and opens a Flipkart search results page for watches for women (and men/couple filters applied).

* --headless: Runs Chrome without a GUI (i.e., it won’t open a browser window). Useful for automation or servers.
* --disable-gpu: Disables GPU rendering (often needed for compatibility with headless mode)

service = Service(ChromeDriverManager().install())

driver = webdriver.Chrome(service=service, options=options)

url = "[Flipkart Wrist Watch](https://www.flipkart.com/search?q=watches+for+women&sid=r18%2Cf13&as=on&as-show=on&otracker=AS_QueryStore_OrganicAutoSuggest_2_7_na_na_ps&otracker1=AS_QueryStore_OrganicAutoSuggest_2_7_na_na_ps&as-pos=2&as-type=RECENT&suggestionId=watches+for+women%7CWrist+Watches&requestId=459df78d-0a5e-48f0-97b3-58d0b514e43c&as-searchtext=watches&p%5B%5D=facets.ideal_for%255B%255D%3DCouple&p%5B%5D=facets.ideal_for%255B%255D%3DWomen&p%5B%5D=facets.ideal_for%255B%255D%3DMen)”

driver.get(url)

* ChromeDriverManager().install(): Automatically downloads the correct ChromeDriver version.
* webdriver.Chrome(...): Launches a Chrome browser instance using the above options.
* driver.get(url) opens the provided url’s result page in the background.

content = driver.page\_source

soup = BeautifulSoup(content)

* driver.page\_source: Get the full HTML content of the current page loaded by Selenium.
* BeautifulSoup(content): Parse that HTML content using BeautifulSoup so you can easily extract data like product names, prices, ratings, etc., using tags and classes.
* To summarize It converts the webpage into a structured format that’s easier to search and scrape.

watches = soup.find\_all('div',class\_='cPHDOP col-12-12')

* Find all <div> elements in the parsed HTML (soup) that have the class:  
  'cPHDOP col-12-12'

The details scraped from the site are

* Product Name – Name of the product
* Price – Price of the product
* Category - Whether the product is for Men or Women or Couple
* Ratings – Ratings provided for the product by the customer, calculated out of 5
* Number of Reviews – Number of Reviews provided by the customers for the specific product
* Number of Ratings - Number of Ratings provided by the customers.

**Demo(Product Name)**

watch\_names= []

for i in watches:

name = i.find('a',class\_="WKTcLC")

if name:

watch\_names.append(name.get\_text(strip=True))

else :

watch\_names.append('N/A')

* This loop goes through each product block (watches) and tries to find the watch name inside an <a> tag with class "WKTcLC".
* If the name is found, it's stripped and added to watch\_names; otherwise, 'N/A' is added.

**Demo(Product Price)**

watch\_price =[]

for i in watches :

price = i.find('div',class\_ ="Nx9bqj")

if price:

watch\_price.append(price.get\_text(strip=True))

else:

watch\_price.append('N/A')

* This loop searches each product block (watches) for a <div> tag with class "Nx9bqj" that contains the price.
* If the price exists, it adds the cleaned text to watch\_price; if not, it adds 'N/A'.

**Data Scrapping**

base\_url = [Flipkart Wrist Watch](https://www.flipkart.com/search?q=watches+for+women&sid=r18%2Cf13&as=on&as-show=on&otracker=AS_QueryStore_OrganicAutoSuggest_2_7_na_na_ps&otracker1=AS_QueryStore_OrganicAutoSuggest_2_7_na_na_ps&as-pos=2&as-type=RECENT&suggestionId=watches+for+women%7CWrist+Watches&requestId=459df78d-0a5e-48f0-97b3-58d0b514e43c&as-searchtext=watches&p%5B%5D=facets.ideal_for%255B%255D%3DCouple&p%5B%5D=facets.ideal_for%255B%255D%3DWomen&p%5B%5D=facets.ideal_for%255B%255D%3DMen)

This is the base URL from the renowned e-commerce site after applying filters – Men, Women and Couple.

num\_pages = 150

watch\_names = []

watch\_prices = []

watch\_categories = []

watch\_ratings = []

watch\_reviews = []

The number of pages chosen is 150 and new lists are created for each field for the values to be stored.

for page\_num in range(1, num\_pages + 1):

print(f"Scraping page {page\_num}...")

* It loops from page 1 to 150, to gather data from 150 pages.
* The following code showcases which page is being scrapped and will also provide an error message that the certain page scrapping has failed if that is the case.

url = f"{base\_url}&page={page\_num}"

* For each page, a new URL is formed with &page=1, &page=2, etc.

response = requests.get(url)

soup = BeautifulSoup(response.content, 'html.parser')

* Makes an HTTP request to Flipkart.
* Parses the HTML content using BeautifulSoup.

watches = soup.find\_all('div', class\_='cPHDOP col-12-12')

* As we have already seen in the demo - this code finds all the products under the container 'cPHDOP col-12-12'

**Name,Price, Category Scrapping**

name\_tag = watch.find('a', class\_="WKTcLC")

price\_tag = watch.find('div', class\_="Nx9bqj")

category\_tag = watch.find('div', class\_="SDsN9S")

* These are the details available in the front page thus the class name is looped through the available items from “watches”, scrapped and stored in the empty lists which was created before.
* If these tags are not found, default to 'N/A'.

for watch in watches:

# Product name( can be replaced with price\_tag, category\_tag)

name\_tag = watch.find('a', class\_="WKTcLC") # can be changed as per the field we want to scrape

if name\_tag:

product\_name = name\_tag.get\_text(strip=True)

else:

product\_name = 'N/A'

The above code’s class and the name can be changed as per our needs.

**Rating, Review Scrapping**

product\_link\_tag = watch.find('a', class\_="WKTcLC")

if product\_link\_tag:

product\_link = "https://www.flipkart.com" + product\_link\_tag['href']

product\_response = requests.get(product\_link)

if product\_response.status\_code == 200:

product\_soup = BeautifulSoup(product\_response.content, 'html.parser')

* Because of the rating and reviews not being available in the front page, the href has to be accessed in order to get the product features link for which we are again using **requests.get(product\_link)**

rating\_tag = product\_soup.find('div', class\_="XQDdHH \_1Quie7")

reviews\_tag = product\_soup.find('span', class\_="Wphh3N")

* Once again using the list of items stored in watches we are getting the rating provided to the product and also the number of reviews and ratings provided to the product.
* If not available, it defaults to 'No Ratings' or 'No Reviews'.

watch\_names.append(product\_name)

watch\_prices.append(product\_price)

watch\_categories.append(category)

watch\_ratings.append(ratings)

watch\_reviews.append(reviews)

We will add all the extracted fields into the respective lists and consequently the N/A, No Ratings and No Reviews are also added to the list if that is the result.

data = {'Name': watch\_names,

'Price': watch\_prices,

'Category': watch\_categories, 'Rating': watch\_ratings, 'Reviews': watch\_reviews}

The final data dictionary is converted so that we can extract it into a csv file which will be used for Machine Learning models.

df = pd.DataFrame(data)

df.to\_csv('watches\_data.csv', index=False)

We will create a dataframe using Pandas and also convert it into a CSV file.

**Data Cleaning/Preprocessing**

df.replace('N/A', np.nan, inplace=True)

df.dropna(subset=['Name'], inplace=True)

df.reset\_index(drop=True, inplace=True)

* This code will change all the N/A values to NaN which represents null value.
* This code drops all the rows where the Name is a Null Value as it cannot be filled.
* Index all the available rows for identification and to get the total count of rows.

First Preprocessing done was to identify the null values from the fields. There were two null values one from Reviews and the other one from Number of Ratings.

df['Reviews'] = df['Reviews'].fillna(method = 'bfill')

df['Number of Ratings'] = df['Number of Ratings'].fillna(method = 'bfill')

Both of these column’s null values were replaced using backward fill.

The total number of rows we have is 6 column and the number of rows is 210 from 0 to 209 and so the total number of values are 1260. Datatype of each and every column is provided below.

0 Name 210 non-null object

1 Price 210 non-null object

2 Category 210 non-null object

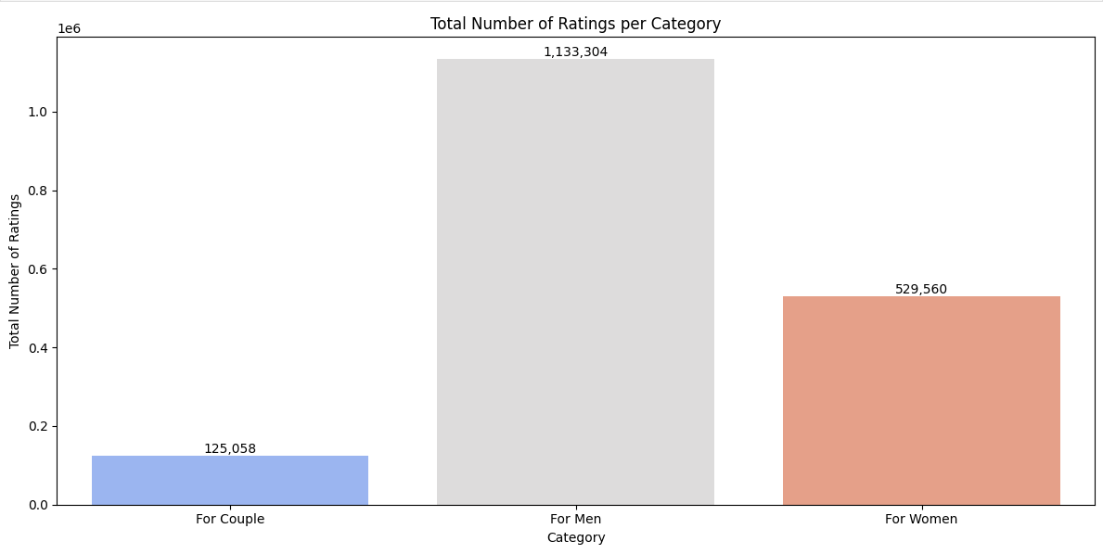
3 Rating 210 non-null float64

4 Reviews 210 non-null float64

5 Number of Ratings 210 non-null float64

**EDA**

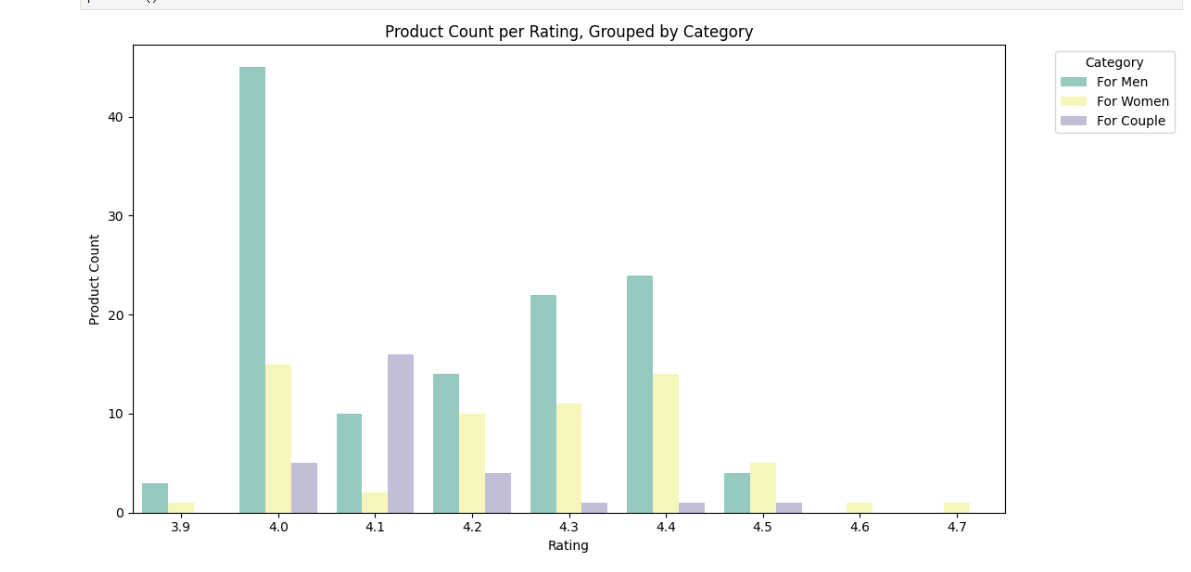
**Univariate Analysis**

****

**Insights:**

1. Watch products specialized for Men have the highest number of ratings in total compared to the other category
2. Products sold for Women have the second highest however it is less than 50% of the ratings received by the Men Specialized watches
3. Watches sold specifically for Couple had the least number of ratings in total which is just over 10% of what was received by Watches sold for Men and compared to Women Watch it is 20%
4. The total number of ratings received for Men product is 1,133,304, For Women it was 529,560 and for Couple it as 125,058 totalling the number of ratings to 17,87,922.

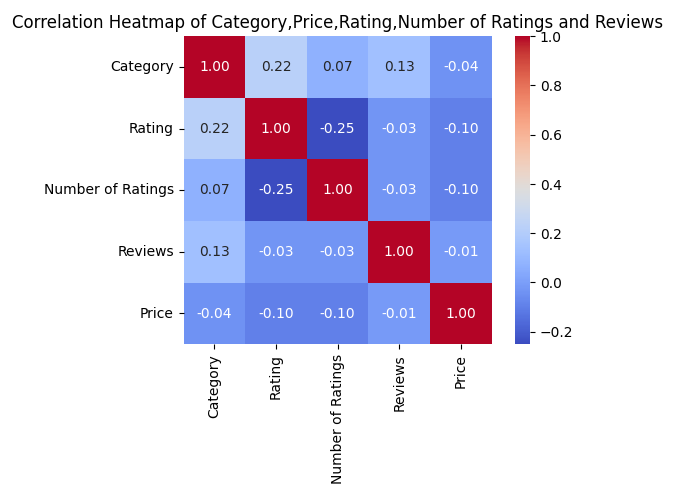
**Bivariate Analysis**

****

**Insights:**

1. The highest count of average rating received by any category is 4 by Men and the same goes for Women
2. 4.1 is the highest average rating received by Watches that is sold specifically for Couple
3. Apart from 4.1, 4.5, 4.6,4.7 every other rating has the category Men in their highest
4. There are no ratings from Men and Couple category which has the average rating of 4.6 and 4.7
5. Women category watches have ratings provided in all rating level out of 5
6. In the average rating of 4.5 Women watches has the highest number of ratings compared to two other categories
7. 3.9 rating average has only categories from Men and Women and not Couple.

**Multivariate Analysis**

****

**Insights:**

1. Strongest relationship is between Category and Rating
2. Weakest relationship is between Number of Ratings and Rating
3. There are three positive relationship and five weak relationships
4. Category and most of fields except Price has the strongest relationship
5. Price has the weakest relationship with other fields
6. Reviews have weakest relationship with other fields except Category
7. Number of Ratings have the weakest relationship with other fields except Category.

**Modelling**

label\_encoder = LabelEncoder()

df[df.select\_dtypes(include=['object']).columns] = df.select\_dtypes(include=['object']).apply(label\_encoder.fit\_transform)

In order to insert the data into the model, we are converting all the categorical values into numerical using LabelEncoder Module.

x = df[['Name','Price','Rating','Reviews','Number of Ratings']]

y = df['Category']

We are assigning the x value as everything except Category since it has to be the input predictors list classification, y is the target.

scaler = MinMaxScaler()

scaler.fit(x)

Normalized\_data = scaler.transform(x)

We will normalize the values into a certain range (0,1) using MinMaxScaler.

**K-Means Clustering**

Normalized\_data = pd.DataFrame(Normalized\_data, columns=x.columns)

kmeans = KMeans(n\_clusters=3, random\_state=42)

kmeans.fit(Normalized\_data)

df['Cluster'] = kmeans.labels\_

silhouette\_avg = silhouette\_score(Normalized\_data, kmeans.labels\_)

print(f"Silhouette Score: {silhouette\_avg}")

plt.scatter(Normalized\_data.iloc[:, 0], Normalized\_data.iloc[:, 1], c=kmeans.labels\_, cmap='viridis')

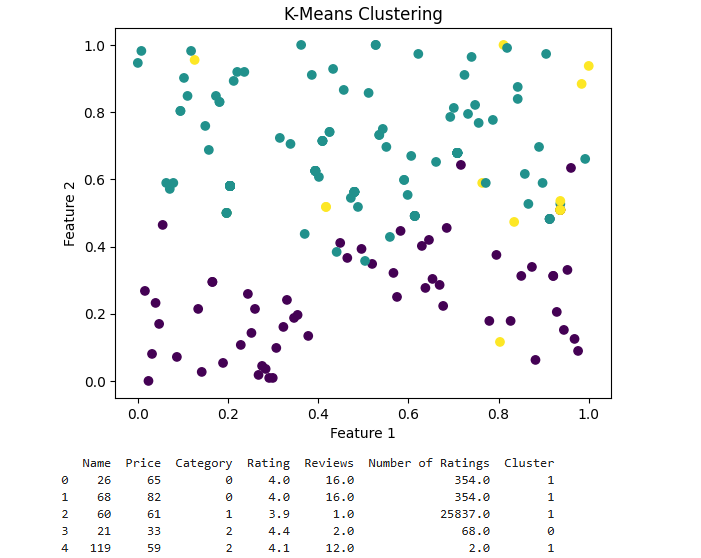
plt.title("K-Means Clustering")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

1. **Data Normalization:**  
   Normalizing data ensures that all features are on a similar scale, preventing any single feature from dominating the clustering process. This is crucial for algorithms like K-Means, which are sensitive to feature magnitudes.
2. **Choosing the Number of Clusters**:  
   The n\_clusters parameter in K-Means determines how many groups the data should be split into. This needs to be specified in advance and can be determined using methods like the Elbow Method or Silhouette Score to find the optimal number.
3. **Cluster Assignment**:  
   After fitting the model, K-Means assigns each data point to the nearest centroid. These assignments are captured in the kmeans.labels\_ attribute and added as a new column in the DataFrame, indicating which cluster each point belongs to.
4. **Silhouette Score for Evaluation**:  
   The Silhouette Score helps evaluate the quality of the clustering. It ranges from -1 to 1, with a higher score indicating better-defined clusters, where points are well-grouped and far from other clusters. Silhouette Score received is 0.26968674287464095
5. **Visualization of Clusters**:  
   Visualizing the clusters, typically using a scatter plot, helps to inspect how well the algorithm has performed. If you have more than two features, dimensionality reduction techniques like **PCA** can be used for plotting.



**Insights:**

**Teal** - Represents popular products with high ratings and a large number of reviews.  
These items likely have good visibility and consistent sales on the platform.

**Purple** - Consists of lower-priced products with very few reviews and ratings.  
Likely represents budget items or newly listed products with limited engagement.

**Yellow** - Contains premium or niche products with high prices but fewer interactions.  
These may be specialty items that appeal to a smaller target audience.

* **Silhouette Score (26%)** - This low score indicates that the clusters are not well separated, and some points may be poorly assigned. It suggests that the clustering structure is weak and could be improved with better features or a different algorithm.
* **Hopkins Statistic (20%)** - A value this low means the dataset has little to no natural cluster tendency. It implies that the data may be close to random in structure, making clustering less effective.

cluster\_labels = kmeans.predict(Normalized\_data)

df['Cluster'] = cluster\_labels

We will be adding a new column to our dataset to indicate each product’s cluster membership as per the instructions.

1. **Logistic Regression**

classifier = LogisticRegression()

classifier.fit(x\_train,y\_train)

We are using Logistic Regression as our first model to test the accuracy score for prediction, the model has been imported from sklearn library which can be found in the libraries used section.

We will use x train and y train data to train the model and use x test and y test to predict the accuracy score of the model.

x\_train\_prediction = classifier.predict(x\_train)

training\_data\_accuracy = accuracy\_score(y\_train, x\_train\_prediction)

We will train the accuracy score of the training data using x train and y train data.

**Accuracy score of training: 0.6071428571428571**

x\_test\_prediction = classifier.predict(x\_test)

testing\_data\_accuracy = accuracy\_score(y\_test,x\_test\_prediction)

We are using the test data for testing data accuracy and the result is provided below.

**Accuracy score of Logistic Regression Testing before Hyper Parameter tuning: 0.5952380952380952**

**Hyper Parameter Tuning Process**

log\_reg = LogisticRegression(max\_iter=1000)

param\_grid = {'C': [0.01, 0.1, 1, 10, 100],

'penalty': ['l1', 'l2'],

'solver': ['liblinear']}

grid\_search = GridSearchCV(log\_reg, param\_grid, cv=5, scoring='accuracy')

grid\_search.fit(x\_train, y\_train)

print("Best Parameters:", grid\_search.best\_params\_)

best\_model = grid\_search.best\_estimator\_

y\_prediction\_log = best\_model.predict(x\_test)

print("Test Accuracy:", accuracy\_score(y\_test, y\_prediction\_log))

1. Searches for the best logistic regression hyperparameters using grid search with cv = 5
2. Trains the logistic regression model using the best-found parameters on the training data.
3. Predicts on the test data and prints the accuracy and best hyperparameter combination.

**The Testing score accuracy of Logistic regression has increased to 64%**

1. **Support Vector Machine – Classifier**

best\_classifier=SVC(kernel='linear',random\_state=0)

best\_classifier.fit(x\_train,y\_train)

Next to Logistic Regression we are using SVM – Classifier to see the accuracy score for the same dataset. We are using the training data for the classifier first.

y\_pred=best\_classifier.predict(x\_test)

We will get the accuracy score by using the testing data and predicting the data.

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

print('accuracy score: ',accuracy)

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

print('confusion matrix: ',cm)

Now using the testing data and the prediction we are calculating the accuracy score.

**The accuracy score received for SVM-C before hyperparameter tuning is 0.6190476190476191**

**Hyper Parameter Tuning Process**

print(best\_classifier.get\_params())

param\_grid = {

'C': [0.1, 1, 10, 100], # Regularization parameter

'kernel': ['linear', 'rbf'], # Kernel types: linear and radial basis function (RBF)

'gamma': ['scale', 'auto'] # Gamma values for 'rbf' kernel}

grid\_search = GridSearchCV(estimator=SVC(), param\_grid=param\_grid, cv=10, n\_jobs=-1, verbose=4)

grid\_search.fit(x\_train, y\_train)

best\_classifier = grid\_search.best\_estimator\_

y\_pred = best\_classifier.predict(x\_test)

final\_accuracy\_SVC = accuracy\_score(y\_test, y\_pred)

print("Final Accuracy on SVC Test Set:", final\_accuracy\_SVC)

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

print('Confusion Matrix:\n', cm)

1. The grid search explores various combinations of the regularization parameter (C), kernel types (linear, rbf), and gamma values (scale, auto) to find the best set of parameters for the SVC model.
2. It uses 10-fold cross-validation (cv=10) to evaluate the model's performance more reliably, running in parallel with n\_jobs=-1 to speed up the process.
3. After identifying the best parameters, it evaluates the final model on the test set, printing the accuracy score and the confusion matrix for deeper insights into the model's performance.

**The testing score accuracy of SVM – Classifier has increased to 64%**

1. **KNN – Classifier**

clf = KNN(n\_neighbors = 5, metric='euclidean') #here use k=5

clf.fit(x\_train, y\_train)

test\_predict = clf.predict(x\_test)

k\_1 = f1\_score(y\_test, test\_predict, average='weighted') # or 'macro', 'micro'

print("F1 Score:", k\_1)

def Elbow(K):

test\_error = []

for i in K:

clf = KNN(n\_neighbors=i) # Use alias KNN here

clf.fit(x\_train, y\_train)

tmp = clf.predict(x\_test)

f1 = f1\_score(y\_test, tmp, average='weighted') # Multiclass-safe

error = 1 - f1

test\_error.append(error)

return test\_error

k = range(2, 20, 2)

test = Elbow(k)

plt.plot(k, test, marker='o')

plt.xlabel("Number of Neighbors (K)")

plt.ylabel("1 - F1 Score (Error)")

plt.title("Elbow Method using F1 Score")

plt.grid(True)

plt.show()

clf = KNN(n\_neighbors = 8) #after find K-value by elbow method

clf.fit(x\_train, y\_train)

test\_predict = clf.predict(x\_test,)

k\_2 = f1\_score(y\_test, test\_predict, average='weighted'

1. **Initial KNN Model**:  
   We have imported a KNN classifier with 5 neighbors and fit the model to the training data. It then calculates the F1 score on the test set using a weighted average.
2. **F1 Score Calculation**:  
   The initial F1 score (k\_1) is printed, showing the model's performance before tuning the k value.
3. **Elbow Method**:  
   The Elbow() function evaluates different values of k (2 to 20, with a step of 2), computing the error (1 - F1 score) for each. The goal is to find the k value that minimizes the error.
4. **Elbow Plot**:  
   The error values for different k values are plotted to visualize the "elbow," helping to select the optimal k value for better performance.
5. **Final Model Evaluation**:  
   After selecting the optimal k (e.g., k=7), the model is retrained and its F1 score (k\_2) on the test set is printed, showing the improvement after tuning.

**The accuracy score of KNN Model before Parameter tuning is 0.737037037037037**

**Hyper-parameter tuning Process**

knn\_model = KNN()

param\_grid = {

'n\_neighbors': list(range(1, 21)),

'weights': ['uniform', 'distance'],

'metric': ['euclidean', 'manhattan']}

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

grid\_search.fit(x\_train, y\_train)

print("Best Parameters:", grid\_search.best\_params\_)

print("Best Cross-Validated Accuracy:", grid\_search.best\_score\_)

**best\_knn = grid\_search.best\_estimator\_**

**y\_pred\_knn = best\_knn.predict(x\_test)**

**print("\nClassification Report:\n", classification\_report(y\_test, y\_pred\_knn))**

**final\_accuracy\_knn = accuracy\_score(y\_test, test\_predict)**

**print("Final Accuracy on Test Set:", final\_accuracy\_knn)**

The grid search explores three hyperparameters: n\_neighbors (1 to 20), weights (uniform or distance), and metric (Euclidean or Manhattan) to find the best combination for the model.

A 5-fold cross-validation is used with GridSearchCV to evaluate the model on different subsets of the training data, ensuring robust performance.

After finding the best parameters, the best hyperparameters and cross-validated accuracy are displayed, followed by the classification report and final test accuracy.

**The testing score accuracy of KNN – Classifier has increased to 74%**

1. **Random Forest Classifier**

clf=RandomForestClassifier(n\_estimators=100,random\_state=2)

clf.fit(x\_train,y\_train)

y\_pred\_RF=clf.prediczt(x\_test)

from sklearn import metrics

print("Accuracy:",metrics.accuracy\_score(y\_test, y\_pred\_RF))

**The accuracy score received for Random Forest before Parameter tuning is 0.6666666666666666**

**Hyper Parameter Tuning Process**

rf\_model = RandomForestClassifier(random\_state=2)

param\_grid = {

'n\_estimators': [100, 200, 300, 500],

'max\_depth': [None, 10, 20, 30],

'min\_samples\_split': [2, 5, 10],

'min\_samples\_leaf': [1, 2, 4],

'max\_features': ['auto', 'sqrt', 'log2'],

'bootstrap': [True, False],

'criterion': ['gini', 'entropy']}

grid\_search = GridSearchCV(estimator=rf\_model, param\_grid=param\_grid, cv=4, scoring='accuracy', verbose=4, n\_jobs=-1)

grid\_search.fit(x\_train, y\_train)

print("Best Parameters:", grid\_search.best\_params\_)

print("Best Cross-Validated Accuracy:", grid\_search.best\_score\_)

best\_rf\_model = grid\_search.best\_estimator\_

y\_pred\_rf = best\_rf\_model.predict(x\_test)

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred\_rf))

final\_accuracy\_rf = accuracy\_score(y\_test, y\_pred\_rf)

print("Final Accuracy on Test Set:", final\_accuracy\_rf)

The grid search explores multiple hyperparameters of the Random Forest model, including the number of trees, max depth, split criteria, and feature selection strategies to find the best-performing configuration.

It uses 3-fold cross-validation with GridSearchCV to evaluate each hyperparameter combination based on accuracy, running in parallel to speed up the process.

After fitting, it outputs the best parameters, evaluates the model on the test set, and prints the classification report and final test accuracy.

**The testing score accuracy of Random Forest has increased to 69%**

1. **XGBoost Classifier**

xgb\_cfl = xgb.XGBClassifier(n\_jobs = -1)

xgb\_cfl.get\_params()

xgb\_cfl.fit(x\_train, y\_train) # default

xgb\_predictions = xgb\_cfl.predict(x\_test)

print(xgb\_predictions)

xgb\_predictions\_prob = xgb\_cfl.predict\_proba(x\_test)

print(xgb\_predictions\_prob)

acc=accuracy\_score(y\_test, xgb\_predictions)

print(acc)

**The accuracy score received for XGboost Classifier before Parameter tuning was 0.7142857142857143**

**Hyper Parameter Tuning Process**

**params = {** 'n\_estimators' : [100, 200, 500, 750], # no of trees

'learning\_rate' : [0.01, 0.02, 0.05, 0.1, 0.25], # eta

'min\_child\_weight': [1, 5, 7, 10],

'gamma': [0.1, 0.5, 1, 1.5, 5],

'subsample': [0.6, 0.8, 1.0],

'colsample\_bytree': [0.6, 0.8, 1.0],

'max\_depth': [3, 4, 5, 10, 12]}

folds = 5

param\_comb = 100

random\_search = RandomizedSearchCV(xgb\_cfl, param\_distributions=params, n\_iter=param\_comb, scoring='accuracy', n\_jobs=-1, cv=5, verbose=3, random\_state=42)

random\_search.fit(x\_train, y\_train)

print('\n Best accuracy for %d-fold search with %d parameter combinations:' % (folds, param\_comb))

print(random\_search.best\_score\_ )

The code performs a **randomized search** on hyperparameters for an XGBoost model (xgb\_cfl) using 5-fold cross-validation (cv=5). It randomly tests 100 combinations of hyperparameters like n\_estimators, learning\_rate, and max\_depth to find the best-performing configuration.

The RandomizedSearchCV is set to optimize for accuracy and runs in parallel with n\_jobs=-1 for faster execution. After fitting the model, it prints the best accuracy score found during the search along with the number of folds and parameter combinations tested.

**The testing score accuracy of XGBoost after tuning is 73%**

**Conclusion**

* The best model with highest accuracy is KNN with the accurate nearest neighbour value with accuracy score of 74% followed by XGBoost – Classifier accuracy score after hyper parameter tuning – 73%
* All the models have accuracy score of almost 60% even before Hyper-Parameter tuning thus explaining the classification models works well with this dataset
* Stakeholders should focus on more products from Couple and Women category considering the Men category shopping will not go down
* Additional information such as returns, review% could be helpful in understanding in-depth situations so that more analysis can be done in order to increase the sales of the specific product
* Analog watches are sold more in quantity compared to digital watches indicating the people buying them are not teens
* There are few items which comes under gift category i.e. provided to couples as gift which should be increased and available for all occasions.