May 2023

Mobile Price Prediction

Pattern Recognition IT352

Ola Ayman Hamed 20200328 S5
Gamal Mohamed Gamal 20200123 S3
Shrook Mohsen Samir 20200253 S4
Braa AbdElhafeiz MohamedSalih 20200815 S3
Almonzir Ali Babkir 20200849 S6
Sara Tarek Abdelalem 20201087 S1

Introduction

The prediction of mobile device prices has become a crucial task in the fast-paced mobile market. To tackle this challenge, machine learning techniques have emerged as powerful tools for analyzing data and building predictive models. In this professional report, we present a comprehensive methodology for mobile price prediction using multiple classifiers. Our goal is to determine the best classifier that provides the highest accuracy for predicting mobile prices.

The report begins with an overview of the mobile market landscape, highlighting the factors that influence mobile pricing, such as specifications. Understanding these factors is essential for developing effective machine learning models for price prediction.

Next, we delve into the methodology for solving the mobile price prediction problem. We describe the data collection process, which involves sourcing diverse and reliable datasets containing information about mobile devices and their corresponding prices. We then preprocess the data, handling missing values, performing feature engineering, and ensuring that the dataset is suitable for training and evaluating the classifiers.

To determine the best classifier for mobile price prediction, we explore and implement multiple classifiers. In this report, we focus on four popular classifiers: Random Forest Classifier, Gaussian NB Classifier, KNN Classifier, and SVM Classifier. Each classifier is explained in detail, including its underlying principles and how it is applied to the mobile price prediction problem.

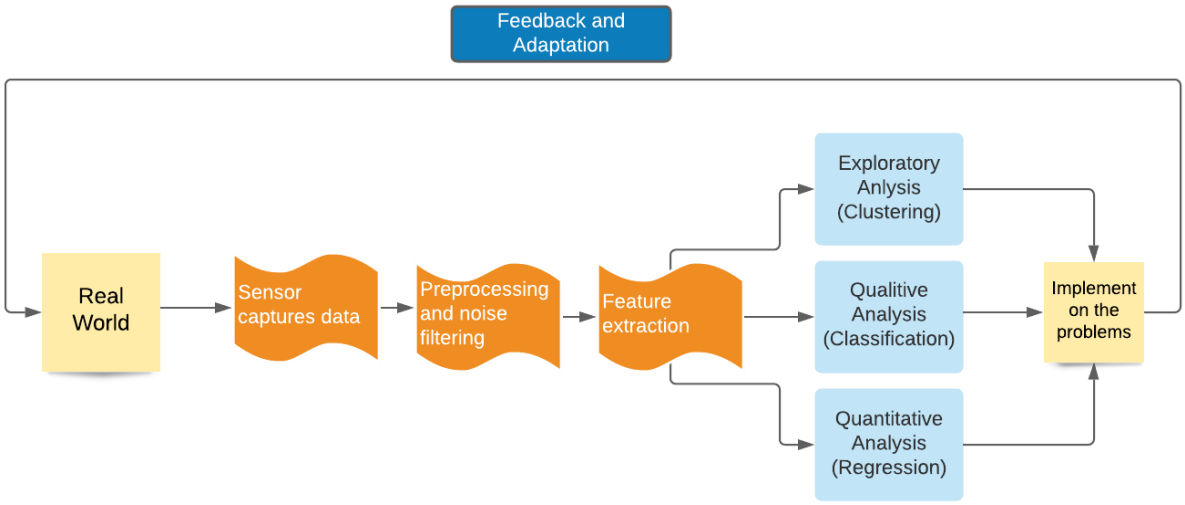
We train and evaluate each classifier using appropriate evaluation metrics, such as accuracy, precision, recall, and F1-score. The performance of each classifier is compared, allowing us to determine which classifier achieves the highest accuracy in predicting mobile prices. The evaluation process includes hyperparameter tuning to optimize the performance of each classifier.

Through our experiments, we identify the best classifier for mobile price prediction based on its accuracy. The results of our analysis provide valuable insights into which classifier performs optimally for this specific problem. These findings have practical implications for businesses in the mobile industry, enabling

them to make informed decisions related to pricing strategies, product positioning, and market competition.

The Problem

We have 2 files that are pre collected and processed, this helped in passing phase 1 “sensors” and phase 2 “preprocessing and enhancement” and then phase 3 “feature extraction.”



The feature extraction phase returned the following features:

- Battery power measured in milliampere-hours (mAh)
- Presence or absence of Bluetooth functionality
- Clock speed of the microprocessor
- Dual SIM support of the phone
- Megapixels of the front camera
- Availability of 4G support
- Internal memory capacity in gigabytes (GB)
- Depth of the mobile device in centimeters (cm)
- Weight of the mobile phone
- Number of cores in the processor
- Megapixels of the primary camera
- Height of pixel resolution
- Width of pixel resolution

- Random-access memory (RAM) capacity in megabytes (MB)
- Height of the mobile screen in centimeters (cm)
- Width of the mobile screen in centimeters (cm)
- Longest duration after a single charge
- Presence or absence of 3G connectivity
- Availability of touch screen functionality
- Availability of Wi-Fi connectivity

Methodology

Reading CSV files

```
trainData=pd.read_csv('train.csv')
testData=pd.read_csv('test.csv')
```

The `trainData` is the data used to train our model, It contains the mobile data and its price range, the prices themselves are not continuous values but discrete categories, the `testData` doesn't contain the `price_range` column, as this is the piece of info that we need to obtain using our model

Analyze the info in the train data file to know what we are dealing with.

```
print(trainData.head())
print(testData.head())
```

```
PS F:\pattern> python .\main.py
   battery_power  blue  clock_speed  dual_sim  fc  four_g  int_memory  m_dep  mobile_wt  n_cores  pc  px_height  px_width   ram  sc_h  sc_w  talk_time  three_g  touch_screen  wifi  price_range
0            842     0          2.2         0   1       0           7    0.6        188        2   2         20       756  2549     9     7         19        0             0     1            1
1           1021     1          0.5         1   0       1          53    0.7        136        3   6        905      1988  2631    17     3          7        1             1     0            2
2            563     1          0.5         1   2       1          41    0.9        145        5   6       1263      1716  2603    11     2          9        1             1     0            2
3            615     1          2.5         0   0       0          10    0.8        131        6   9       1216      1786  2769    16     8         11        1             0     0            2
4           1821     1          1.2         0  13       1          44    0.6        141        2  14       1208      1212  1411     8     2         15        1             1     0            1
   id  battery_power  blue  clock_speed  dual_sim  fc  four_g  int_memory  m_dep  mobile_wt  n_cores  pc  px_height  px_width   ram  sc_h  sc_w  talk_time  three_g  touch_screen  wifi
0   1           1043     1          1.8         1  14       0           5    0.1        193        3  16       226      1412  3476    12     7          2        0             1     0
1   2            841     1          0.5         1   4       1          61    0.8        191        5  12       746       857  3895     6     0          7        1             0     0
2   3           1807     1          2.8         0   1       0          27    0.9        186        3   4      1270      1366  2396    17    10         10        0             1     1
3   4           1546     0          0.5         1  18       1          25    0.5         96        8  20       295      1752  3893    10     0          7        1             1     0
4   5           1434     0          1.4         0  11       1          49    0.5        108        6  18       749       810  1773    15     8          7        1             0     1
```

Those are the first fer entries in train and test files, this shows the column data and give un an idea of what to expect in our analysis.

Filter train and test files from empty entries to reduce errors.

```
#applies the Boolean mask to the trainData, filtering out rows where 'sc_w' is zero.
trainDataFiltered = trainData[trainData['sc_w'] != 0]
trainDataFiltered = trainDataFiltered [trainDataFiltered['sc_h'] != 0]
trainDataFiltered = trainDataFiltered [trainDataFiltered['mobile_wt' ] != 0]
trainDataFiltered = trainDataFiltered [trainDataFiltered['m_dep'] != 0 ]
trainDataFiltered = trainDataFiltered [trainDataFiltered['px_width'] != 0 ]
trainDataFiltered = trainDataFiltered [trainDataFiltered['px_height'] != 0 ]


testDataFiltered = testData[testData['sc_w'] != 0]
testDataFiltered =testDataFiltered[testDataFiltered['sc_h'] != 0 ]
testDataFiltered =testDataFiltered[testDataFiltered['mobile_wt' ] != 0  ]
testDataFiltered =testDataFiltered[testDataFiltered['m_dep'] != 0 ]
testDataFiltered =testDataFiltered[testDataFiltered['px_width'] != 0 ]
testDataFiltered =testDataFiltered[testDataFiltered['px_height'] != 0  ]
```

```
dataframe dimentions before filtering (2000, 21)
dataframe dimentions after filtering (1819, 21)
```

Using filtering to delete entries that contain empty/corrupted data, these parameters can never be zero, that's why exclude the zeros, this will increase accuracy.

Training and validation using train data

```
X=trainDataFiltered.drop(['price_range'], axis=1) #input feature matrix
y=trainDataFiltered['price_range'] # creates the target variable array
Xtst=testDataFiltered.drop(['id'], axis=1)

#train_test_split(Input feature matrix, target variable array, proportion of the data to be allocated to
the validation set, random seed for reproducibility.)
# 20% of the data will be used for validation.
X_train, X_valid, y_train, y_valid= train_test_split(X, y, test_size=0.2, random_state=7)
#  The training set contains 80% of the original data, and the validation set contains 20% of the
original data.
```

preparing the data for machine learning tasks. It creates the input feature matrix X and the target variable array y, creating the feature matrix involves dropping the target column that we wish to compute through our model, and putting this target column in the y variable.

Now we split the data into training and validation sets, we split the input feature matrix X and the target value Y, and we select the value of the `test_size` which is the proportion of the data to be allocated to the validation set. In this case, it is set to 0.2, indicating that 20% of the data will be used for validation, finally the `random_state` Sets the random seed for reproducibility. It ensures that the split is deterministic and consistent across different runs.

The function returns four sets of data: `X_train` (training input features), `X_valid` (validation input features), `y_train` (training target variable), and `y_valid` (validation target variable), The training set contains 80% of the original data, and the validation set contains 20% of the original data.

The main difference between **training data** and **validation data** lies in their purpose and usage. Training data is used to teach the model and adjust its parameters, while validation data is used to assess the model's performance and make decisions about its configuration.

By evaluating the model on unseen validation data, you can gain insights into its generalization capabilities and make adjustments to improve its performance before applying it to real-world scenarios or testing it on a separate test set.

Building confusion matrix

A confusion matrix is a table that summarizes the performance of a classification model by showing the counts of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions. It is a widely used tool for evaluating the performance of machine learning models, particularly in binary classification problems. Each row of the confusion matrix represents the actual class labels, while each column represents the predicted class labels.

- True Positive (TP): The number of samples that were correctly predicted as positive.
- True Negative (TN): The number of samples that were correctly predicted as negative.
- False Positive (FP): The number of samples that were incorrectly predicted as positive (Type I error).

- False Negative (FN): The number of samples that were incorrectly predicted as negative (Type II error).

The confusion matrix provides valuable information about the model's performance, including:

- **Accuracy**: The overall accuracy of the model, (TP + TN) / (TP + TN + FP + FN).
- **Precision**: The proportion of correctly predicted positive samples among all samples predicted as positive, TP / (TP + FP).
- **Recall** (also known as **Sensitivity** or **True Positive Rate**): The proportion of correctly predicted positive samples among all actual positive samples, TP / (TP + FN).
- **Specificity** (also known as **True Negative Rate**): The proportion of correctly predicted negative samples among all actual negative samples, TN / (TN + FP).
- **F1-score**: A harmonic mean of precision and recall, 2 * (Precision * Recall) / (Precision + Recall).

The confusion matrix helps in understanding the model's performance in terms of correctly and incorrectly classified samples. It is particularly useful in situations where the costs or implications of false positives and false negatives are different.

```
def confusionMX(y_test, y_pred, plt_title):
    CMX=confusion_matrix(y_test, y_pred) #compares the actual target values with the predicted target
    values
    print(classification_report(y_test, y_pred))
    sns.heatmap(CMX, annot=True, fmt='g', cbar=False, cmap='BuPu')
    plt.xlabel('Predicted Values')
    plt.ylabel('Actual Values')
    plt.title(plt_title)
    plt.show()
    return CMX
```

The function takes 3 parameters (The actual target values from the test set, The predicted target values obtained from the model, plot title), the confusion matrix is calculated using the built in `confusion_matrix(y_test, y_pred)` from `scikit-learn`.

Then in the console we print the `classification_report`, which provides a summary of various metrics such as precision, recall, F1-score, and support for

each class, also We can represent the confusion matrix using a heatmap from seaborn.

Testing classifiers

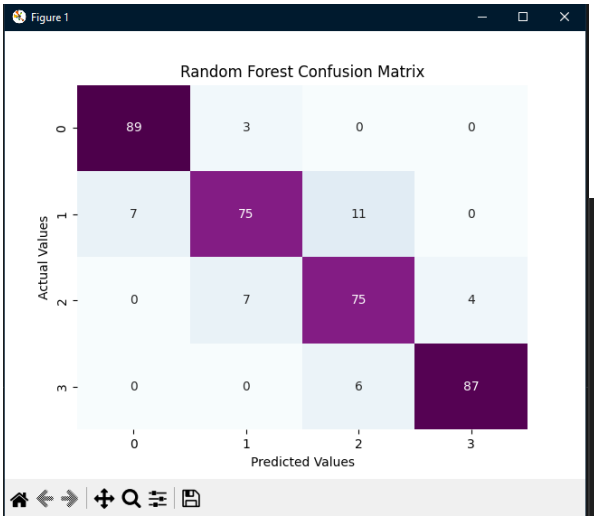
Random Forest Classifier

```python
#-------------------------Random Forest Classifier--------------------------------
def RFV():
    from sklearn.ensemble import RandomForestClassifier

    rfc=RandomForestClassifier()
    rfc.fit(X_train, y_train) #train
    y_pred_rfc=rfc.predict(X_valid)

    print('Random Forest Classifier Accuracy Score: ',accuracy_score(y_valid,y_pred_rfc))
    cm_rfc=confusionMX(y_valid, y_pred_rfc, 'Random Forest Confusion Matrix')

RFV()
```




```
PS F:\pattern> python .\main.py
Random Forest Classifier Accuracy Score:  0.8956043956043956
              precision    recall  f1-score   support

           0       0.93      0.97      0.95        92
           1       0.88      0.81      0.84        93
           2       0.82      0.87      0.84        86
           3       0.96      0.94      0.95        93

    accuracy                           0.90       364
   macro avg       0.90      0.90      0.89       364
weighted avg       0.90      0.90      0.90       364
```

Imagine you have a tough decision to make, and you need some advice. Instead of asking just one person, you decide to ask a group of people with different backgrounds and experiences. Each person gives their opinion, and you make your decision based on the majority opinion. This is like how a Random Forest Classifier works.

In a Random Forest Classifier, instead of one person, you have many decision trees working together as a team. Each decision tree is like a person who gives their own prediction based on a subset of the data. The trees are created by randomly selecting different samples from the dataset and considering only a few random

features at each step. This randomness helps to ensure that each tree learns something different.

Once all the trees have made their predictions, the Random Forest Classifier combines their opinions to make the final prediction. For example, if you're trying to classify something into different categories, each tree would vote for a particular category. The category with the most votes becomes the final predicted category.

Think of a Random Forest Classifier as a group of decision trees working together to help you make predictions. They combine their opinions, take the majority vote, and provide reliable results.

Here the Random Forest Classifier Accuracy Score is 0.8956043956043956

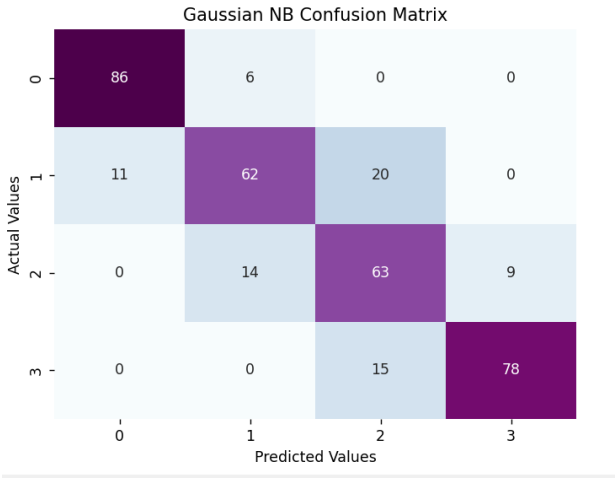
Gaussian NB classifier

```
#---------------------------Gaussian NB classifier-------------------------
def gnb():
    from sklearn.naive_bayes import GaussianNB
    gnb = GaussianNB()

    gnb.fit(X_train, y_train)
    y_pred_gnb=gnb.predict(X_valid)

    print('Gaussian NB Classifier Accuracy Score: ',accuracy_score(y_valid,y_pred_gnb))
    cm_rfc=confusionMX(y_valid, y_pred_gnb, 'Gaussian NB Confusion Matrix')

gnb()
#-----------------------------------KNN Classifier-------------------------------------
```




```
PS F:\pattern> python .\main.py
Gaussian NB Classifier Accuracy Score:  0.7939560439560439
              precision    recall  f1-score   support

           0       0.89      0.93      0.91        92
           1       0.76      0.67      0.71        93
           2       0.64      0.73      0.68        86
           3       0.90      0.84      0.87        93

    accuracy                           0.79       364
   macro avg       0.80      0.79      0.79       364
weighted avg       0.80      0.79      0.79       364
```

The Gaussian NB Classifier is like a clever detective who looks at the features of different mobile phones and tries to figure out the most likely price range for each phone. It uses a special mathematical technique called Bayes' theorem to make its predictions.

Here's how it works: First, the Gaussian NB Classifier learns from a training set of mobile phones with known features and price ranges. It looks at the data and calculates the probability distribution of each price range given the different features. For example, it figures out how likely it is for a phone with high battery power, a great camera, and lots of memory to be in a certain price range.

Once the classifier has learned from the training data, it becomes a detective ready to solve mysteries. When you present it with a new mobile phone with an unknown price range, it looks at the features and calculates the probabilities again. It compares the likelihood of the phone falling into different price ranges based on what it learned from the training data.

The Gaussian NB Classifier assumes that the features are independent and follow a Gaussian (normal) distribution. This means that it assumes each feature, like battery power or camera quality, is not influenced by the others and follows a bell-shaped curve. This simplifies the calculations and allows the classifier to make predictions quickly.

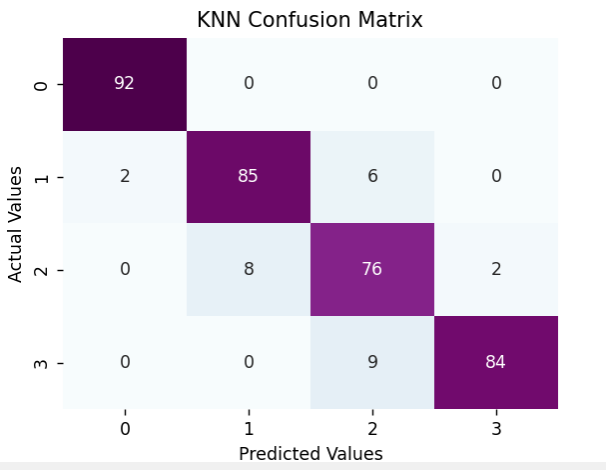
Here, the Gaussian NB Classifier Accuracy Score is 0.7939560439560439

KNN Classifier

```
def knn():
    from sklearn.neighbors import KNeighborsClassifier
    knn = KNeighborsClassifier(n_neighbors=3,leaf_size=25)

    knn.fit(X_train, y_train)
    y_pred_knn=knn.predict(X_valid)

    print('KNN Classifier Accuracy Score: ',accuracy_score(y_valid,y_pred_knn))
    cm_rfc=confusionMX(y_valid, y_pred_knn, 'KNN Confusion Matrix')
```




```
PS F:\pattern> python .\main.py
KNN Classifier Accuracy Score:  0.9258241758241759
              precision    recall  f1-score   support

           0       0.98      1.00      0.99        92
           1       0.91      0.91      0.91        93
           2       0.84      0.88      0.86        86
           3       0.98      0.90      0.94        93

    accuracy                           0.93       364
   macro avg       0.93      0.93      0.93       364
weighted avg       0.93      0.93      0.93       364
```

The KNN Classifier is like a friendly neighbor who looks at the features of different mobile phones and tries to find the most similar phones to the one you want to predict. It then looks at the price ranges of those similar phones and uses them to make its prediction.

Here's how it works: First, the KNN Classifier learns from a training set of mobile phones with known features and price ranges. It stores this information in its memory. When you present it with a new mobile phone, it looks for the "k" closest neighbors from the training set based on the similarity of their features.

To determine the similarity, the KNN Classifier calculates the distance between the features of the new phone and the features of the training set phones. The most common distance measure used is the Euclidean distance, which is like measuring the straight-line distance between two points.

Once it has found the "k" closest neighbors, the KNN Classifier looks at their price ranges. It takes a majority vote or averages the price ranges of those neighbors and assigns that as the predicted price range for the new phone.

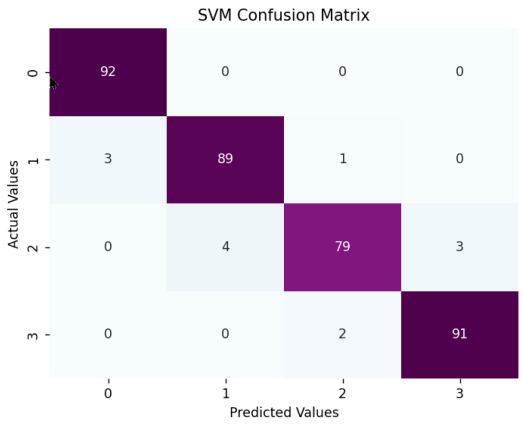
The value of "k" in the KNN Classifier represents the number of neighbors to consider. A larger "k" value means more neighbors are considered, which can lead to a more stable prediction, but it may also introduce more noise from dissimilar phones.

Here, the KNN Classifier Accuracy Score is 0.9258241758241759

SVM Classifier

```python
def SVM(flag):
    from sklearn import svm
    svm_clf = svm.SVC(decision_function_shape='ovo')

    svm_clf.fit(X_train, y_train)
    if flag==0: #train
        y_pred_svm=svm_clf.predict(X_valid)
        print('SVM Classifier Accuracy Score: ',accuracy_score(y_valid,y_pred_svm))
        cm_rfc=confusionMX(y_valid, y_pred_svm, 'SVM Confusion Matrix')
    else:
        y_pred_svm=svm_clf.predict(Xtst)
        for i in range(len(y_pred_svm)):
            print(y_pred_svm[i])
        snsGraphstrain(1, y_pred_svm)
```




```
PS F:\pattern> python .\main.py
SVM Classifier Accuracy Score:  0.9642857142857143
              precision    recall  f1-score   support

           0       0.97      1.00      0.98        92
           1       0.96      0.96      0.96        93
           2       0.96      0.92      0.94        86
           3       0.97      0.98      0.97        93

    accuracy                           0.96       364
   macro avg       0.96      0.96      0.96       364
weighted avg       0.96      0.96      0.96       364
```

The SVM Classifier is like a smart separator that draws a boundary between different groups of mobile phones based on their features. It tries to find the best line or curve that can separate phones in different price ranges most effectively.

Here's how it works: First, the SVM Classifier learns from a training set of mobile phones with known features and price ranges. It looks at the data and tries to find the optimal decision boundary that maximizes the margin, or the space, between different price ranges. This decision boundary is like a line or curve that separates phones of different price ranges as clearly as possible.

The SVM Classifier does this by mapping the features of the mobile phones into a higher-dimensional space. In this higher-dimensional space, it searches for the best decision boundary that can separate the phones effectively. This mapping is

done using a technique called the kernel function, which transforms the data into a more suitable form for separation.

Once the classifier has learned from the training data, it becomes a smart separator ready to make predictions. When you present it with a new mobile phone, it looks at its features and determines which side of the decision boundary it falls into. Based on this, it assigns the corresponding price range to the phone.

The SVM Classifier is especially good at handling complex data and finding non-linear decision boundaries. It can handle cases where a simple straight line cannot effectively separate different price ranges.

Here, the SVM Classifier Accuracy Score is 0.9642857142857143

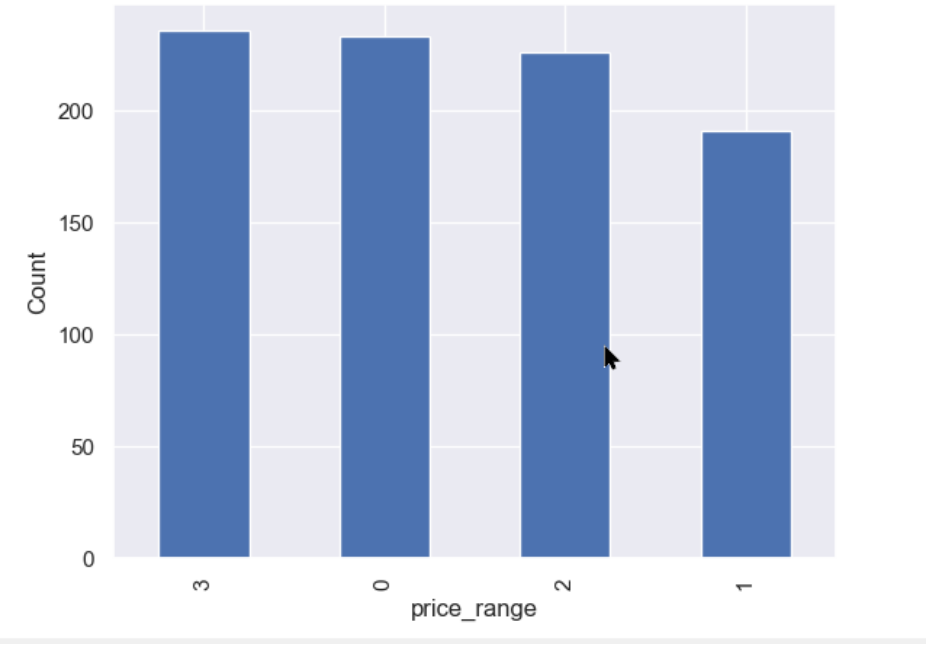
Choosing best classifier

classifier	Accuracy score
SVM	0. 9642857142857143
knn	0.9258241758241759
Random Forest Classifier	0.8956043956043956
Gaussian NB	0.7939560439560439

By far the best classifier that suits our case is the SVM

Testing the chosen classifier

By testing, the output of SVM is array that contains the y values of each entry, by collecting this new data and using seaborn for visualizing the output we get this result.



The bar graph is computed using the following code:

```
else:
    y_pred_svm_series = pd.Series(y_pred_svm) #convert it to a pandas Series before calling the
    value_counts()
    sns.set()
    price_plot = y_pred_svm_series.value_counts().plot(kind='bar')
    plt.xlabel('price_range')
    plt.ylabel('Count')
    plt.show()
```

Conclusion

Selecting an appropriate classifier is of utmost importance in developing the best machine learning model. The choice of classifier significantly influences the model's performance, accuracy, interpretability, and overall success in solving a particular problem.

Firstly, different classifiers are designed to handle specific types of data and patterns. For instance, linear classifiers like Logistic Regression work well when the relationship between input features and output is linear, while decision tree-based classifiers like Random Forests excel in capturing non-linear relationships.

Secondly, the selection of an appropriate classifier impacts the interpretability of the model. Some classifiers, such as Naive Bayes or Decision Trees, provide easily

interpretable rules or probabilities, allowing us to gain insights into the factors driving predictions. On the other hand, complex models like Support Vector Machines or Neural Networks may sacrifice interpretability for improved performance. The nature of the problem and the stakeholders' requirements should guide the choice of a classifier that strikes the right balance between accuracy and interpretability.

Furthermore, the computational and memory requirements of different classifiers vary significantly. For large datasets or limited computing resources, selecting a classifier that is computationally efficient and can handle high-dimensional data is crucial to ensure scalability and feasibility of the model.

In conclusion, the selection of an appropriate classifier plays a pivotal role in building the best machine learning model. It impacts the model's performance, accuracy, interpretability, and computational efficiency, enabling us to derive meaningful insights and make accurate predictions for a given problem domain.