M2 Evidence 2 - Using a framework

Karla Stefania Cruz Muñiz A01661547

Selected algorithm: Random Forest

Random Forest is an algorithm for machine learning whose core are decision trees. This algorithm combines the using of multiple decision trees for getting the different results of all of them to end with a better prediction. In cases of classification problems, this algorithm takes the majority vote to give a final result about the classification of some prediction.

Selected dataset

For this part of the evidence I choose a dataset for finding a mobile price, it looks like this:

battery_pow	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
842	0	2.2	0	1	0	7	0.6	188	2	2	20	756	2549	9	7	19	0	0	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
563	1	0.5	1	2	1	41	0.9	145	5	6	1263	1716	2603	11	2	9	1	1	0	2
615	1	2.5	0	0	0	10	0.8	131	6	9	1216	1786	2769	16	8	11	1	0	0	2
1821	1	1.2	0	13	1	44	0.6	141	2	14	1208	1212	1411	8	2	15	1	1	0	1
1859	0	0.5	1	3	0	22	0.7	164	1	7	1004	1654	1067	17	1	10	1	0	0	1
1821	0	1.7	0	4	1	10	0.8	139	8	10	381	1018	3220	13	8	18	1	0	1	3
1954	0	0.5	1	0	0	24	0.8	187	4	0	512	1149	700	16	3	5	1	1	1	0
1445	1	0.5	0	0	0	53	0.7	174	7	14	386	836	1099	17	1	20	1	0	0	0
509	1	0.6	1	2	1	9	0.1	93	5	15	1137	1224	513	19	10	12	1	0	0	0
769	1	2.9	1	0	0	9	0.1	182	5	1	248	874	3946	5	2	7	0	0	0	3
1520	1	2.2	0	5	1	33	0.5	177	8	18	151	1005	3826	14	9	13	1	1	1	3
1815	0	2.8	0	2	0	33	0.6	159	4	17	607	748	1482	18	0	2	1	0	0	1
803	1	2.1	0	7	0	17	1	198	4	11	344	1440	2680	7	1	4	1	0	1	2
1866	0	0.5	0	13	1	52	0.7	185	1	17	356	563	373	14	9	3	1	0	1	0
775	0	1	0	3	0	46	0.7	159	2	16	862	1864	568	17	15	11	1	1	1	0
838	0	0.5	0	1	1	13	0.1	196	8	4	984	1850	3554	10	9	19	1	0	1	3
595	0	0.9	1	7	1	23	0.1	121	3	17	441	810	3752	10	2	18	1	1	0	3
1131	1	0.5	1	11	0	49	0.6	101	5	18	658	878	1835	19	13	16	1	1	0	1
682	1	0.5	0	4	0	19	1	121	4	11	902	1064	2337	11	1	18	0	1	1	1
772	0	1.1	1	12	0	39	0.8	81	7	14	1314	1854	2819	17	15	3	1	1	0	3
1709	1	2.1	0	1	0	13	1	156	2	2	974	1385	3283	17	1	15	1	0	0	3
1949	0	2.6	1	4	0	47	0.3	199	4	7	407	822	1433	11	5	20	0	0	1	1
1602	1	2.8	1	4	1	38	0.7	114	3	20	466	788	1037	8	7	20	1	0	0	0

As we can see, the dataset contains a lot of relevant information for classifying a mobile phone into a prince range indicating how high the price is. There are 4 possible ranges, 0, 1, 2 and 3.

You can find the dataset in the next kaggle link:
https://www.kaggle.com/datasets/iabhishekofficial/mobile-price-classification/data
I used the training part because it's where the results are shown and I can have supervised learning. You can find this csv as *'train.csv'*.

Code explanation

I'm using pandas for data set managging. In this case, the next line is where the data set is imported for using it along the code.

```
# Load the training data
train_df = pd.read_csv('train.csv')
```

This second part is where I distinct the input variable from the target variable. For this dataset we need to see if the code classifies correctly in terms of price of the given mobile phone, so we can say Y is the target variable and X is the rest of the dataset as the input data. It's also important to mention in the *drop* function, we use as an argument the axis = 1, meaning a column is being dropped.

```
# Separate input data (X) from the target data (y)
X = train_df.drop('price_range', axis=1)
y = train_df['price_range']
```

Next we have the separation between the training and validation set.
train_test_split: Splits the dataset into 80% training data and 20% validation data.

```
# Split the training set into training and validation (80% - 20%)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)
```

When dividing this dataset I selected the 20% for validation because it is a commonly used value in practice ensuring the model is trained with enough data while validating with a representative set.

Consecutive line is the creation of the Random Forest model using scikit-learn. It is imported from sklearn.ensemble because Random Forest is an ensemble model, which means predictions of multiple models, in this case trees, are combined and used for making a decision.

```
# Create the Random Forest model
model = RandomForestClassifier(random_state=40, n_estimators=300, max_depth=17)
```

The *max_depth* find as hyperparameter in the function above was decided by having multiple tests with different values and selecting the one with better performance, as shown here:

max_depth = 10

```
Average accuracy with cross-validation: 0.8749934276250065
```

max_depth = 15

```
Average accuracy with cross-validation: 0.8825034614508299
```

max_depth = 17

```
Average accuracy with cross-validation: 0.8885149937781517
```

max_depth = 20

```
Average accuracy with cross-validation: 0.883015230383651 4
```

max_depth = 18

```
Average accuracy with cross-validation: 0.8850097271149904
```

Also, the *random_state* and *n_estimators* were tested, the first one taking values like 30, 40, 42 and 50; and the second one taking the values 100, 200, 300 and 400.

Fit is used for training a model with given data.

```
# Train the model with the training data
model.fit(X_train, y_train)
```

Here we make a prediction using the Random Forest model. This prediction is saved in *y_pred*, also, this prediction is made on *x_val* which is validation data.

```
# Make predictions on the validation set
y_pred = model.predict(X_val)
```

As mentioned above, in this part using the *accurancy_score* function which also comes from scikit-learn but from the metrics module. This function works by providing the real value and the predicted value, taking both values the function calculates the percentage of correct predictions.

```
# Calculate the accuracy
accuracy = accuracy_score(y_val, y_pred)
print("Accuracy in validation data:", accuracy)
```

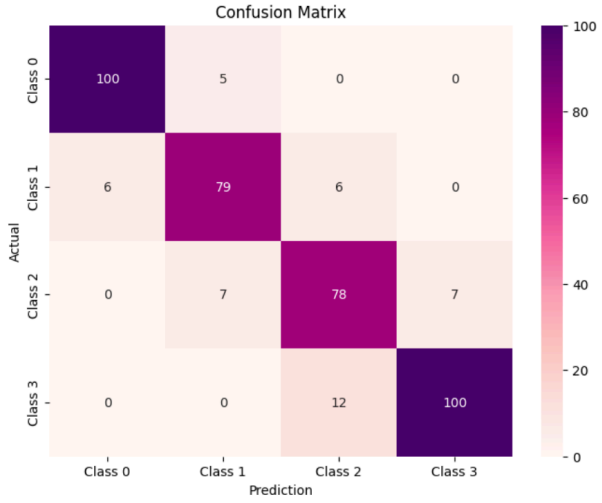
For the next code part, the *confusion_matrix* function is used for obtaining how many predictions were correct or incorrect for each class, because as we mentioned previously, we have four possible classes.

```
# Print confusion matrix
conf_matrix = confusion_matrix(y_val, y_pred)
print("Confusion Matrix:")
print(conf_matrix)
```

The result is printed in the terminal, but if we have the possibility and the library installed for matplotlib, we can uncomment the next part of the code:

```
# Plot confusion matrix
# plt.figure(figsize=(8, 6))
# sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='RdPu',
#             xticklabels=['Class 0', 'Class 1', 'Class 2', 'Class 3'],
#             yticklabels=['Class 0', 'Class 1', 'Class 2', 'Class 3'])
# plt.xlabel('Prediction')
# plt.ylabel('Actual')
# plt.title('Confusion Matrix')
# plt.show()
```

This commented part will plot the confusion matrix as shown in the next image:



In this image of the confusion matrix we can see how the model classifies each of the tests done for the different classes. As we can see, for class 0 we got 105 cases where 100 were classified correctly and 5 were classified incorrectly as class 1. That's how we can read this confusion matrix, showing how many cases were classified incorrectly.

The next part is a classification report also using scikit-learn metrics. This function provides detailed performance metrics for each class, including precision, recall, and F1-score.

```
# Classification report
class_report = classification_report(y_val, y_pred)
print("Classification Report:")
print(class_report)
```

Also as part of metrics we have the ROC-AUC for measuring how well the model separates the classes. It evaluates the ability of a model to distinguish between classes, also providing a measure of the performance of all classification thresholds, as we know, as closer as the value is from 1, the better the model distinguishes classes. For this function I had to use *ovr*, because the results can be multiple classes.

```
# Calculate ROC-AUC for each class
roc_auc = roc_auc_score(y_val, model.predict_proba(X_val), multi_class='ovr')
print("ROC-AUC:", roc_auc)
```

Getting closer to the end, we have the cross validation, this is going to help us to ensure the generalizability of the model. The number of splits was selected by testing 5, 6, 7 and 8, obtaining the best performance at 7 by a minimum difference with 5.

```
# Perform cross-validation with 7 folds
cv = StratifiedKFold(n_splits=7, shuffle=True, random_state=42)
accuracy_scores = cross_val_score(model, X, y, cv=cv, scoring='accuracy')
```

Using 7 splits:

```
Average accuracy with cross-validation: 0.89150673887516
```

Using 5 splits:

```
Average accuracy with cross-validation: 0.891
```

When using 8 splits we got: 0.884.
In the same image, we have *accuracy_scores*, where cross validation runs and evaluates the model's accuracy for each fold, saving in the variable the scores for each fold.
As another parameter we have *shuffle*; in this case it is used for shuffling the data before dividing it in folds, ensuring every fold is a representative "mini data set".
The last parameter I would like to explain from this part of the code is the parameter *scoring*, where I specify the accuracy is gonna be the metric for evaluating the performance of the model during the cross validation.

Saving the accuracy allows us to get the mean, showing what's the average accuracy using cross validation as shown in the next image.

```
# Calculate the average accuracy in the folds
average_accuracy = accuracy_scores.mean()
print("\nAverage accuracy with cross-validation:", average_accuracy)
```

Last, in the code, we have the analysis of the final importance for each feature in the dataset, basically, how important each feature is in making predictions. This works by taking the *feature_importances_* attribute and sorting them so we can see them in order of importance.

```
# Feature Importance Analysis
importances = model.feature_importances_
indices = importances.argsort()[::-1]
features = X.columns

print("\nFeature importance:")
for i in range(X.shape[1]):
    print(f"- {features[indices[i]]}: {importances[indices[i]]}")
```

Output Analysis

```
Accuracy in validation data: 0.8925

Confusion Matrix:
[[100   5   0   0]
 [  6  79   6   0]
 [  0   7  78   7]
 [  0   0  12 100]]

Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.95      0.95       105
           1       0.87      0.87      0.87        91
           2       0.81      0.85      0.83        92
           3       0.93      0.89      0.91       112

    accuracy                           0.89       400
   macro avg       0.89      0.89      0.89       400
weighted avg       0.89      0.89      0.89       400

ROC-AUC: 0.9846028582758193

Average accuracy with cross-validation: 0.89150673887516

Feature importance:
- ram: 0.4729423253781277
- battery_power: 0.07333625591273821
- px_width: 0.056434484129174706
- px_height: 0.05629527056577619
- mobile_wt: 0.040560124435432124
- int_memory: 0.038464440645029956
- talk_time: 0.03044114245389165
- pc: 0.03029454746259559
- sc_w: 0.02859986035860718
- clock_speed: 0.02828604453120346
```

← Accuracy calculated by predictions giving the samples of the validation dataset.

← Detailed view of how the model is making the classification

← Metrics for each class. Precision, recall and F1-score

← Model's ability to distinguish between classes
← Accuracy using cross validation.

← Importance of each feature for taking the final decision

As we could see, my model works a little bit better classifying classes 0 and 3, and get's a little bit confused in classes 1 and 2.