Portfolio2- Week6- Classification

Qiaoyu Wang

0 Start and find a new dataset

This week we learned Classification, including decision trees, K-Nearest Neighbors (KNN), and various evaluation metrics to a dataset. In order to understand how the process of these models worked, I found a new dataset about the apple quality (from https://www.kaggle.com/datasets/nelgiriyewithana/apple-quality), which was a very clean dataset for exploring classification.

1 Train Models

1.1 Decision Trees

I initiated the experiment by using two features, 'size' and 'weight', to predict the apple quality, and set the parameter 'max_depth' as 3. The result of the accuracy stood at 59.66666666666667, which indicated limited predictive power. I changed different 'max_depth' like 5, 8, 10, but the result of accuracy still fluctuated between 55 to 60. Then I realized that it might because the features I selected were irrelevant, which meant that the size and weight of apples were not related very much to the apple quality.

```
#Pick two features
feature1 = "Size"
feature2 = "Weight"
#Pick max_depth
max_depth = 50
```

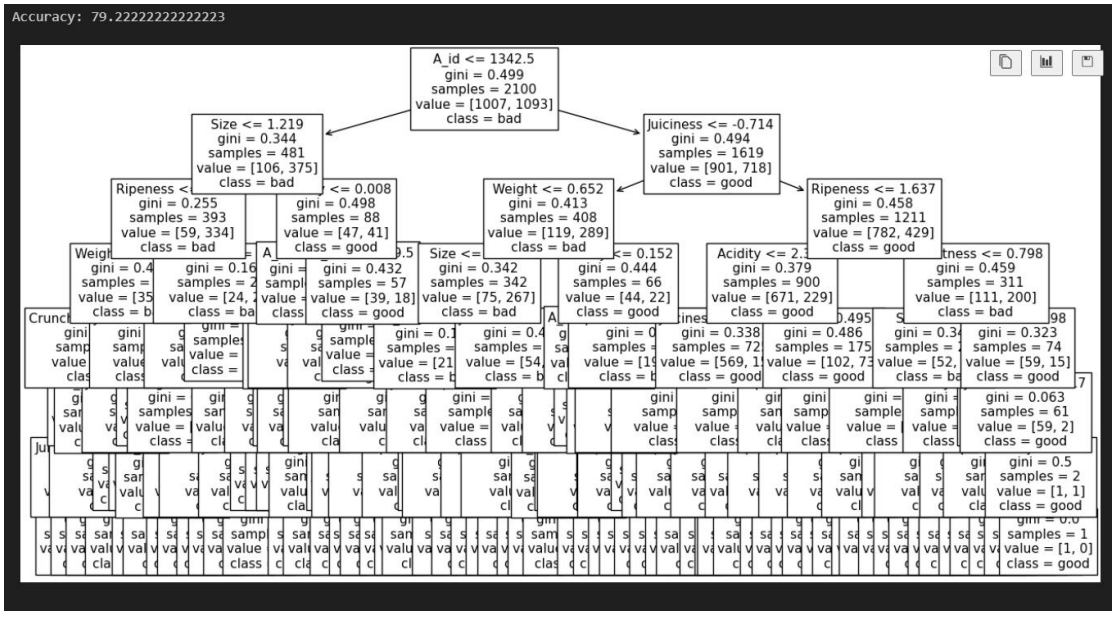
[19] ✓ 0.0s

```
#Train model, plot decision boundary
x = df[[feature1,feature2]].values
y = pd.to_numeric(df["Quality"])
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size=0.3, random_state=0)
model = DecisionTreeClassifier(max_depth=max_depth)
model.fit(x_train,y_train)
#See if the model works
y_pred = model.predict(x_test)
num_incorrect = (y_test != y_pred).sum()
total = y_test.shape[0]
acc = (total - num_incorrect) / total * 100
print("Accuracy:", acc)
```

[20] ✓ 0.0s

```
Accuracy: 56.222222222222214
```

Then I tried to use the whole feature set with the ‘max_depth=3’. To be expected, I got a better accuracy which was 72.33333333333334, but it still wasn’t an ideal number for accuracy. I also changed the ‘max_depth’ to see when the accuracy could reach the maximum. The result was ‘max_depth=7’, which could give a number of accuracy to 79.2. Larger ‘max_depth’ values led to decreased accuracy, suggesting potential over fitting..



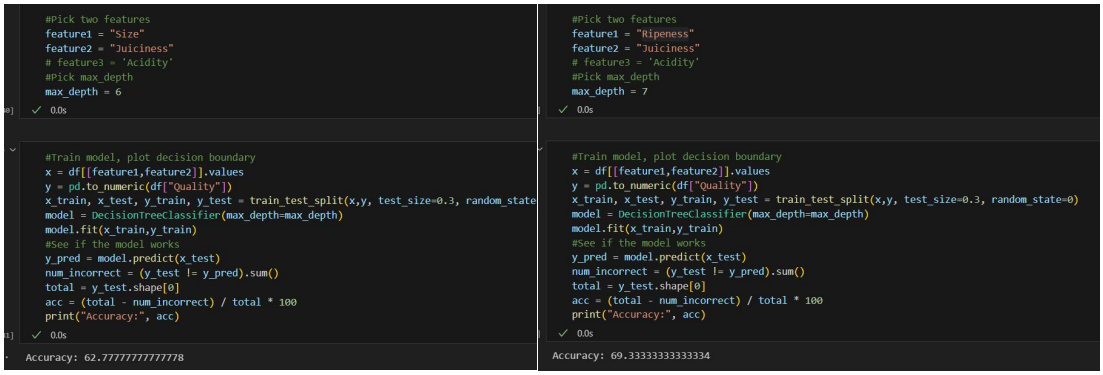
From the above decision tree, I found it difficult to discern how the model favoured some features over others. Consequently, I sought out techniques online and discovered 'feature_importance' as a promising method, which returned a ranking of feature importance.

```
X = df.drop(columns=['Quality'])
y = df['Quality']

# Now you can use X in your code
importances = model.feature_importances_
indices = np.argsort(importances)[::-1]
for f in range(X.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))
```

✓ 0.0s

```
1. feature 0 (0.271023)
2. feature 4 (0.205434)
3. feature 5 (0.193051)
4. feature 2 (0.157359)
5. feature 1 (0.089347)
6. feature 6 (0.056905)
7. feature 3 (0.026880)
```

Now I got the potential importance of each feature and it appeared that the feature ‘Size’ and ‘Juiciness’ might be more relevant to apple quality. However, upon incorporating these two features into the model, I observed an unexpected outcome that **the accuracy dropped from 70, associated with the seemingly less important features 'Ripeness' and 'Juiciness', to 62.**



```
#Pick two features
feature1 = "Size"
feature2 = "Juiciness"
# feature3 = 'Acidity'
#Pick max_depth
max_depth = 6
```

✓ 0.0s

```
#Train model, plot decision boundary
x = df[[feature1,feature2]].values
y = pd.to_numeric(df["Quality"])
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size=0.3, random_state
model = DecisionTreeClassifier(max_depth=max_depth)
model.fit(x_train,y_train)
#See if the model works
y_pred = model.predict(x_test)
num_incorrect = (y_test != y_pred).sum()
total = y_test.shape[0]
acc = (total - num_incorrect) / total * 100
print("Accuracy:", acc)
```

✓ 0.0s

```
Accuracy: 62.77777777777778
```

```
#Pick two features
feature1 = "Ripeness"
feature2 = "Juiciness"
# feature3 = 'Acidity'
#Pick max_depth
max_depth = 7
```

✓ 0.0s

```
#Train model, plot decision boundary
x = df[[feature1,feature2]].values
y = pd.to_numeric(df["Quality"])
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size=0.3, random_state=0)
model = DecisionTreeClassifier(max_depth=max_depth)
model.fit(x_train,y_train)
#See if the model works
y_pred = model.predict(x_test)
num_incorrect = (y_test != y_pred).sum()
total = y_test.shape[0]
acc = (total - num_incorrect) / total * 100
print("Accuracy:", acc)
```

✓ 0.0s

```
Accuracy: 69.33333333333334
```

This made me quite confused about the 'feature_importance' method, as it seems **more complex than merely identifying the most important features. The interaction between features, randomness in Data and other relevant information may also need to be taken into consideration.**

So how to address this issue?

1. Relying on feature_importance isn't the only way to capture information. Exploring different feature engineering techniques would be necessary to find more connections from the data.

2. I may need to try different combinations of features, such as setting a for_loop of whole features or manually selecting random ones, to identify the most effective feature set for the model.

3. Instead of relying on a single decision tree model, I found an ensemble method called Random Forests, which can naturally handle feature importance and interactions between features.

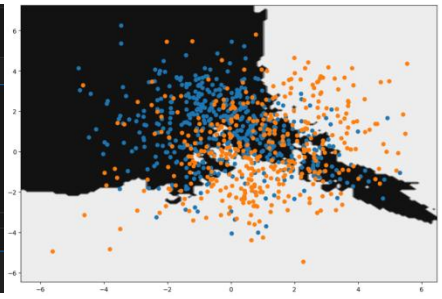
1.2 KNN

I proceeded to train another model KNN, using the most relevant feature set found above, and got the accuracy of approximately 68, which closely matched that of the decision model. Despite altering the feature set, I found that the accuracy of both models remained nearly identical. **Then I tried the whole feature list, and got an unexpected high accuracy of 88, which meant that the KNN model performed very well.** The reason might be KNN model is more appropriate to use when the input data is a small number.

I'm going to use evaluation metrics to compare these 2 models (Chapter 1.3).

Furthermore, I also experimented with various values of k, yet the result didn't show much variation.

```
k = 25
✓ 0.0s

knn_model = KNeighborsClassifier(n_neighbors=k)
knn_model.fit(x_train,y_train)
#See if the model works
y_pred_knn = knn_model.predict(x_test)
num_incorrect_knn = (y_test != y_pred_knn).sum()
total_knn = y_test.shape[0]
acc_knn = (total_knn - num_incorrect_knn) / total_knn * 100
print("Accuracy:", acc_knn)
✓ 0.1s

Accuracy: 68.55555555555556
```

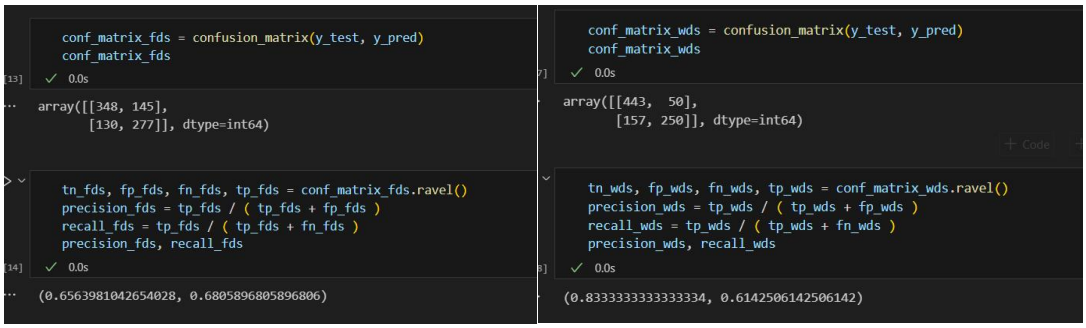


```
k = 20
✓ 0.0s

x_labels = list(df.columns.values)
x_labels.remove("Quality")
x = df[x_labels]
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size=0.3, random_state=0)
knn_model = KNeighborsClassifier(n_neighbors=k)
knn_model.fit(x_train,y_train)
#See if the model works
y_pred_knn = knn_model.predict(x_test)
num_incorrect_knn = (y_test != y_pred_knn).sum()
total_knn = y_test.shape[0]
acc_knn = (total_knn - num_incorrect_knn) / total_knn * 100
print("Accuracy:", acc_knn)
✓ 0.1s

Accuracy: 88.0
```

1.3 Evaluation Metrics

As mentioned before, in order to evaluate both models on the testing set, I employed the confusion matrix, precision and recall.

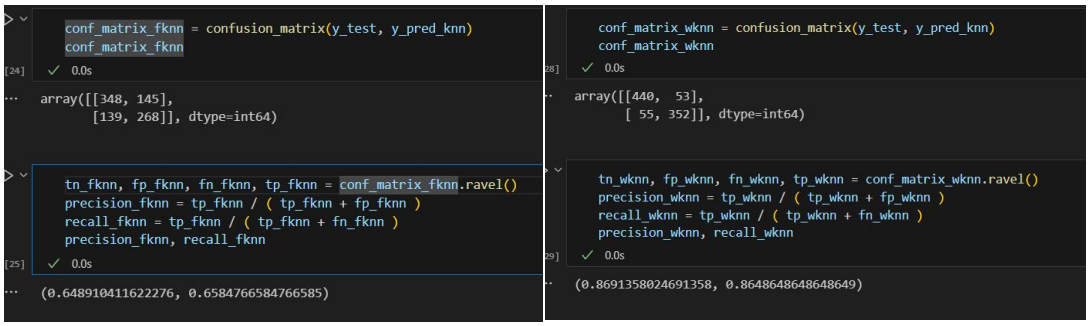
Following figures showed the evaluation of the decision tree model, using different numbers of features. It is evident that the model based on only two selected features exhibited low precision and low recall. The outcome suggested that the model's performance was poor, aligning with the analysis of accuracy. Conversely, the model of whole features showed high precision and low recall, which indicated that the model is very conservative in its predictions, making fewer positive predictions overall but with a high accuracy among those predictions.



(Left: Confusion Matrix, precision, recall of selected 2 features in decision tree model;

Right:Confusion Matrix, precision, recall of whole feature list in decision tree model)

Let's turn a look on the evaluation matrix of the RNN model. Similar to the selected features model in the decision tree, the RNN model also exhibited low precision and low recall. However, when it come to the model with all features, it yielded an impressive desirable result of both high precision and high recall. This outcome signifies that the model not only made accurate predictions but also captured a large portion of the positive instances in the dataset.



(Left: Confusion Matrix, precision, recall of selected 2 features in KNN model;

Right:Confusion Matrix, precision, recall of whole feature list in KNN model)

So how to explain this difference of precision and recall between decision trees model and KNN model?

The inherent characteristics of each model differ significantly.

KNN considers the local neighborhood of data points to make predictions, based on the class labels of its nearest neighbors. It's sensitive to the distribution of data, and can capture most of the positive instances by design. So KNN tends to have high recall, especially when dealing with small and simple datasets.

In contrast, decision trees tend to be prone to overfitting, particularly when the tree depth is not appropriately constrained in a small number of inputs. This may let result in the model mistakenly capturing noise or outliers as part of the decision boundaries, which ultimately leads to lower recall.

2 Conclusion

Both models have their own advantages and disadvantages, and the choice between them depends on different situations.

Normally, when the number of the inputs is simple and limited, KNN often emerges as a good choice. This model boasts simplicity in training and involves fewer parameters, which also indicates that if the datasets go larger, the model will be hard to handle with. Besides, because it's more sensitive to individual data points, it'll easier to be impacted with the noisy data, posing a risk of overfitting. Also its rules of prediction make it difficult to interpret the reasoning behind each prediction.

Conversely, decision trees offer clarity in illustrating the prediction process. Each node in the tree corresponds to a decision based on a feature value, making it easy to understand and find the valued features. However, constructing an optimal decision tree may require exploring

various splits and features, which also seem to be not suitable for larger datasets.

Then new questions come out that **how can we use these two models more effectively to handle larger datasets? Are there alternative techniques that can offer better computational performances than KNN and decision trees for large datasets? How do these alternative methods compare in terms of predictive performance and interpretability?** Further research is needed to delve deeper into these questions.