apple-quality-prediction

March 6, 2024

1 Fruit Quality Classification Exercise using KNN classifiers

In this notebook, we'll go through and explore a fruit goodness dataset, build a few classifiers using standard pre-coded classifier code using python libraries, and then ultimately code up our own from-scratch KNN classifier that we have coded ourselves and compare how it works. Using our own code, we'll look at the k value to see how it effects the quality of the classifier on both the training and test sets. To address this problem, you need to write the code wherever it says **FILLIN** or `**FILLIN**` below.

The code you produce *must* be your own code, that you wrote yourself without help from anything other than your own brain.

1.0.1 First, Load libraries and Data

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.tree import DecisionTreeClassifier


import warnings
# Suppress FutureWarnings (this might not be a good idea as this will mean that␣
 ↪you might miss out on important warnings)
# You can remove this next line if you want to see any warnings that might␣
 ↪occur.
warnings.simplefilter(action='ignore', category=FutureWarning)
```

1.0.2 Load the apple quality data into a pandas dataframe `df` that was provided on the canvas page.

[2]:
```
df = pd.read_csv('apple_quality.csv', encoding='utf-8')
```

1.0.3 We can look at the first few rows of the data as follows

[3]:
```
# Pandas DataFrames have many methods that allow us to explore a dataset we've␣
 ↪read in. As mentioned in class, it is a good idea whenever
# presented with a new dataset to "take the dataset out for a beer.". This␣
 ↪means looking at the data in a variety of ways, including
# simple printing of a few bits of the data as well as computing a few simple␣
 ↪statistics over the data. We do this in the next few cells.
df.head()
```

[3]:

	A_id	Size	Weight	Sweetness	Crunchiness	Juiciness	Ripeness	\
0	0.0	-3.970049	-2.512336	5.346330	-1.012009	1.844900	0.329840	
1	1.0	-1.195217	-2.839257	3.664059	1.588232	0.853286	0.867530	
2	2.0	-0.292024	-1.351282	-1.738429	-0.342616	2.838636	-0.038033	
3	3.0	-0.657196	-2.271627	1.324874	-0.097875	3.637970	-3.413761	
4	4.0	1.364217	-1.296612	-0.384658	-0.553006	3.030874	-1.303849	

	Acidity	Quality
0	-0.491590483	good
1	-0.722809367	good
2	2.621636473	bad
3	0.790723217	good
4	0.501984036	good

1.0.4 Overall info of the DataFrame

We can also find out the number of columns and type of each column.

[4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4001 entries, 0 to 4000
Data columns (total 9 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   A_id         4000 non-null   float64
 1   Size         4000 non-null   float64
 2   Weight       4000 non-null   float64
 3   Sweetness    4000 non-null   float64
 4   Crunchiness  4000 non-null   float64
 5   Juiciness    4000 non-null   float64
 6   Ripeness     4000 non-null   float64
 7   Acidity      4001 non-null   object
```

```
 8   Quality      4000 non-null   object
dtypes: float64(7), object(2)
memory usage: 281.4+ KB
```

1.0.5 Final rows

Lets look at the final set of rows.

[5]:
```
# As we see above, most of the columns are of type float64 (numeric data) which is good for us to build a KNN classifier,
# but then a few are of dtype 'object' which means they are strings. We need to convert these to numeric data before we can use them in our classifier.
# But first, why is the 'Acidity' column not numeric? Maybe we can find out by looking at the bottom of the DataFrame.
df.tail()
```

[5]:

	A_id	Size	Weight	Sweetness	Crunchiness	Juiciness	Ripeness	Acidity	Quality
3996	3996.0	-0.293118	1.949253	-0.204020	-0.640196	0.024523	-1.087900	1.854235285	good
3997	3997.0	-2.634515	-2.138247	-2.440461	0.657223	2.199709	4.763859	-1.334611391	bad
3998	3998.0	-4.008004	-1.779337	2.366397	-0.200329	2.161435	0.214488	-2.229719806	good
3999	3999.0	0.278540	-1.715505	0.121217	-1.154075	1.266677	-0.776571	1.599796456	good
4000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Created_by_Nidula_Elgiriyewithana	NaN

1.0.6 Remove bad rows

We see that the final row has some NaNs as well as attribution text, so we drop the final row in place.

[6]:
```
# Lets drop the final row in place since it has NaNs and a string in it.
df.drop(4000, axis=0, inplace=True)
```

1.0.7 Type conversion

Convert the A_id column to be of type int32

[7]:
```
# Also, the id field was read in as a float, but it is really an integer. We can convert it to an integer type.
df['A_id'] = df['A_id'].astype('int32')
```

1.0.8 Set the index

Next, we set the index of the dataframe to the values in the column named 'A_id'.

[8]:
```
df.set_index('A_id', inplace=True)
```

1.0.9 The 'Quality' column contains our labels (what we will predict). Lets find out how many distinct unique values it contains.

[9]:
```
df['Quality'].unique()
```

[9]:
```
array(['good', 'bad'], dtype=object)
```

1.0.10 Labels all good?

In the above, you should see that the labels are 'good' and 'bad', so this is a fruit classification task into two labels. Our goal is to build a classifier into these two categories. First, we need to change the 'Acidity' feature to be float64 type like the others.

[10]:
```
df['Acidity'] = df['Acidity'].astype('float64')
```

1.0.11 Double check

Lets double check that it all worked.

[11]:
```
df.info()
```

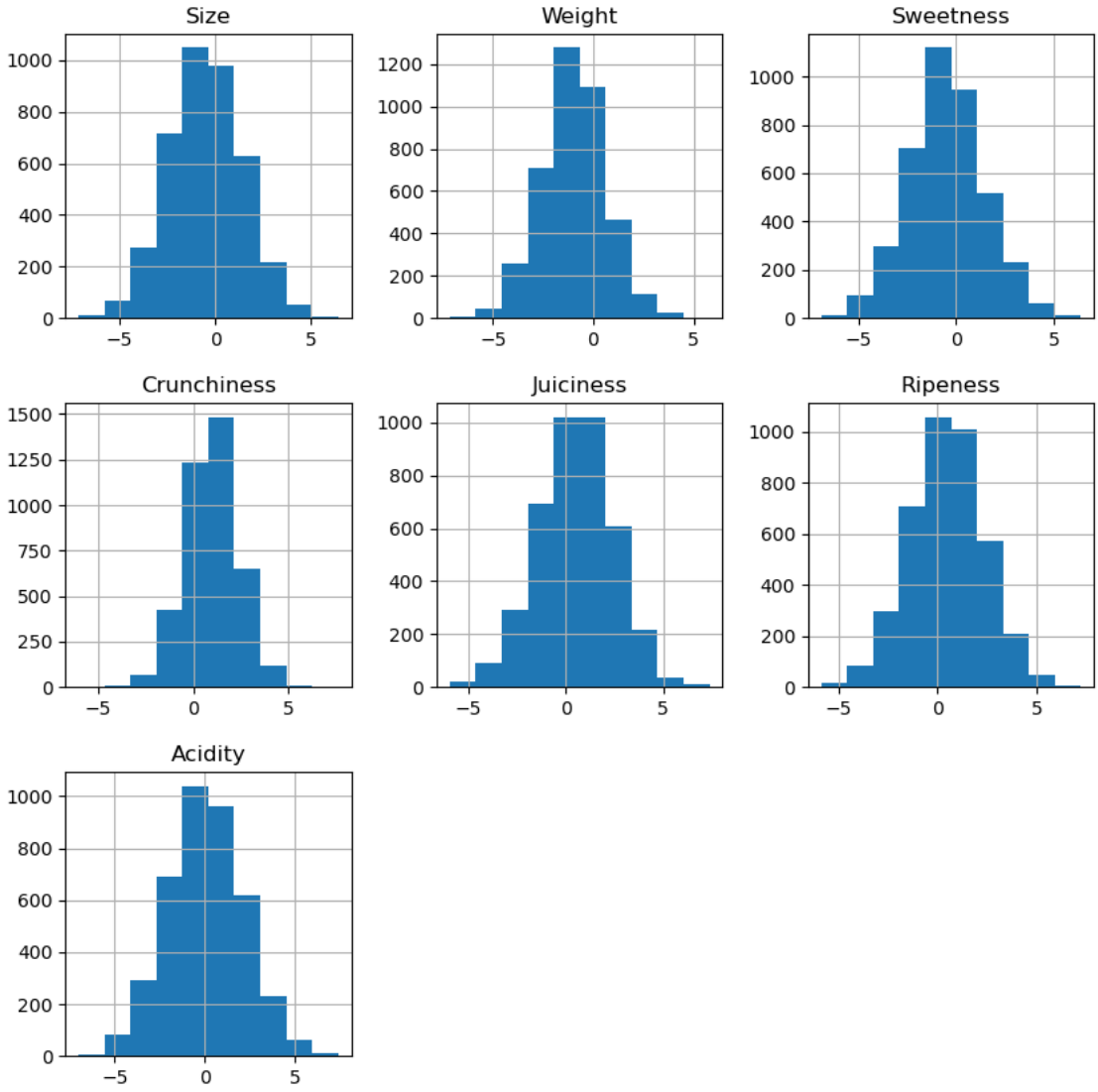
```
<class 'pandas.core.frame.DataFrame'>
Index: 4000 entries, 0 to 3999
Data columns (total 8 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Size         4000 non-null   float64
 1   Weight       4000 non-null   float64
 2   Sweetness    4000 non-null   float64
 3   Crunchiness  4000 non-null   float64
 4   Juiciness    4000 non-null   float64
 5   Ripeness     4000 non-null   float64
 6   Acidity      4000 non-null   float64
 7   Quality      4000 non-null   object
dtypes: float64(7), object(1)
memory usage: 265.6+ KB
```

1.0.12 Another beer

Now, lets buy our data another beer, and get it to reveal itself to us by looking at the histograms of each of the features which is, in general, always an informative thing to do. Note that these are the marginal histograms, i.e., we're looking at the histogram only at each feature individually, we're not yet looking at the joint histograms of any two or more features.

[12]:
```
df.hist(figsize=(10,10))
```

[12]:
```
array([[<Axes: title={'center': 'Size'}>,
        <Axes: title={'center': 'Weight'}>,
        <Axes: title={'center': 'Sweetness'}>],
       [<Axes: title={'center': 'Crunchiness'}>,
        <Axes: title={'center': 'Juiciness'}>,
        <Axes: title={'center': 'Ripeness'}>],
       [<Axes: title={'center': 'Acidity'}>, <Axes: >, <Axes: >]],
      dtype=object)
```



1.0.13 Analysis of the histograms

From the above histograms, and from the perspective of designing our KNN classifier, the data is centered around zero for all features and has similar standard deviation in all of them. Therefore,

this histogram tells us the data is normalised. In kNN, the data being normalised makes calculation easier, which is why this is important. Further, the histogram could help us in identifying a maximum value of "k" (number of nearest neighbours) to be used in our simulations. There is no point in setting a "k" value greater than the maximum number (or even second/third highest number depending on the data) of samples in a bin of any feature in the dataset. Hence, in this case, these histograms could help us limit k down to a value less than 1500

1.0.14 More beer

Next, pandas has many other provisions to get to know our data better, lets 'describe' the data using a variety of statistics, including means, standard deviations, min, max, and various percentiles.

[13]:
```
df.describe()
```

[13]:

	Size	Weight	Sweetness	Crunchiness	Juiciness	Ripeness	Acidity
count	4000.000000	4000.000000	4000.000000	4000.000000	4000.000000	4000.000000	4000.000000
mean	-0.503015	-0.989547	-0.470479	0.985478	0.512118	0.498277	0.076877
std	1.928059	1.602507	1.943441	1.402757	1.930286	1.874427	2.110270
min	-7.151703	-7.149848	-6.894485	-6.055058	-5.961897	-5.864599	-7.010538
25%	-1.816765	-2.011770	-1.738425	0.062764	-0.801286	-0.771677	-1.377424
50%	-0.513703	-0.984736	-0.504758	0.998249	0.534219	0.503445	0.022609
75%	0.805526	0.030976	0.801922	1.894234	1.835976	1.766212	1.510493
max	6.406367	5.790714	6.374916	7.619852	7.364403	7.237837	7.404736

1.0.15 Analysis

From the above set of statistics, we see that all features are in the range (-8,+8), but seem to be centred around 0. Further, the features are concentrated near the mean (which is approximately zero), and so, the data appears normalised. Overall, does this dataset looks ok (i.e., free from errors or bugs) at least from what we can tell so far? Why or why not? Thus far, the dataset looks okay. The count of all features is the same, and the mean, standard deviation, min, max and 25/50/75% points seem reasonable. The data also seems normalised which is very useful in simplying kNN calculations!

1.0.16 Balance

Also, how balanced is the data in terms of 'good' and 'bad'? In general, if a task has an equal number of each category, it is easier to deal with, if a data is very imbalanced then it will be harder to build a classifier for it.

[14]:
```
df['Quality'].value_counts()
```

[14]:
```
Quality
good    2004
bad     1996
Name: count, dtype: int64
```

1.0.17 Analysis of balance

Regarding balance, we see that this data is fairly well balanced on a high level. The number of good and bad samples seems similar.

1.0.18 More visualization

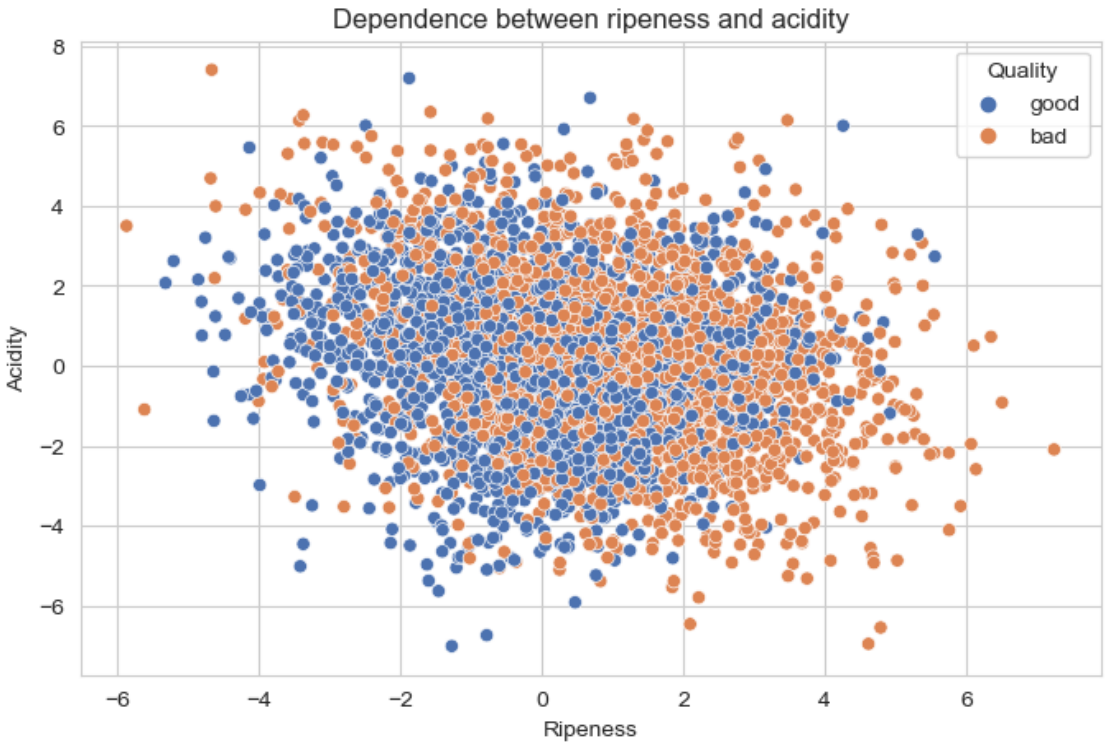
Next, lets do some additional data visualization analyses to, again, better get to know our fruit data.

[15]:
```
sns.set_style('whitegrid')
sns.set_palette('deep')
```

1.0.19 Pairwise Dependencies

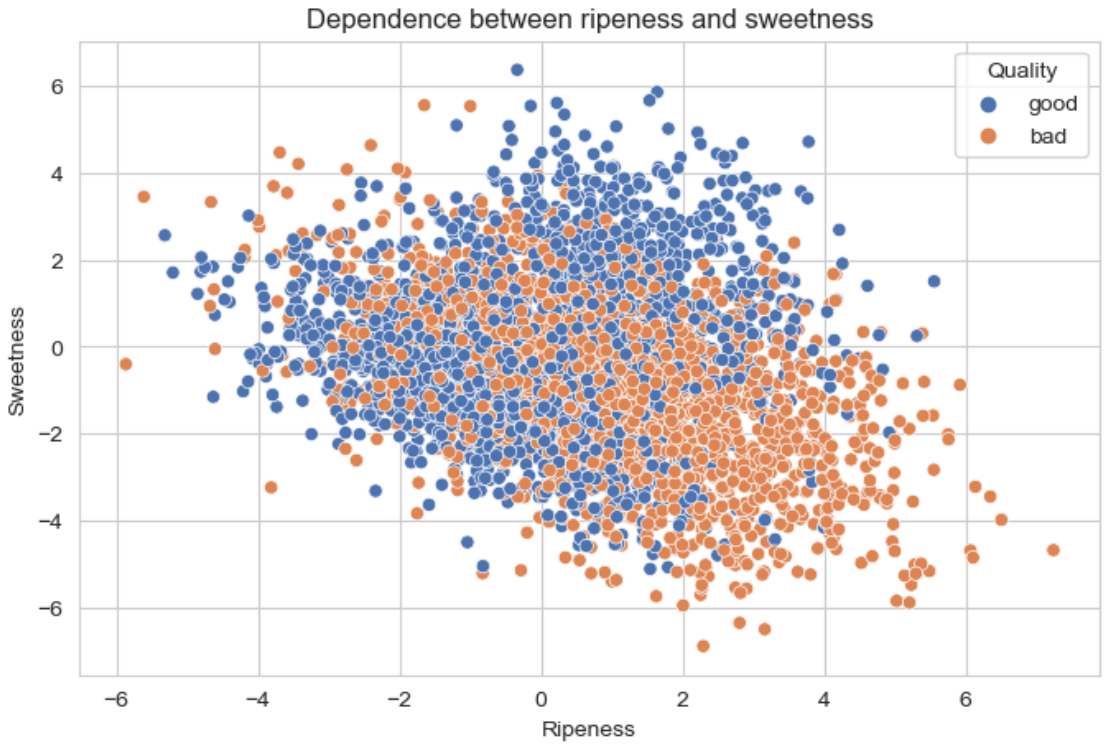
While we live in a 3D world, and we're looking at a 2D screen, what we can do is select some pairs of features and plot 2D scatter plots of them, and then color each dot depending on the class label (in the present case "good" vs. "bad") to see if there are some pairs of features that render the prediction particularly easy. For example, can we find pairs of features such that "good" vs. "bad" is linearly separable?

[16]:
```
plt.figure(figsize=(8, 5))
sns.scatterplot(data=df, x='Ripeness', y='Acidity', hue='Quality')
plt.title("Dependence between ripeness and acidity")
plt.show()
```

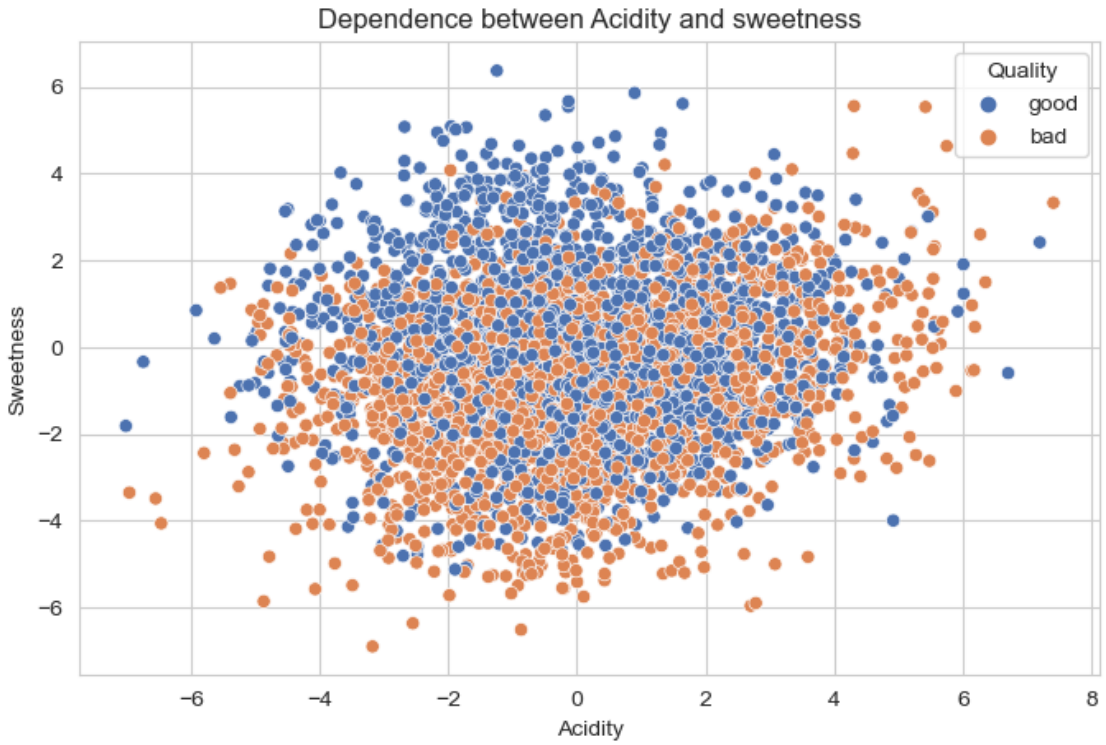


The above shows that looking at 'Ripeness' vs 'Acidity' shows that the data is difficult to separate linearly. That being said, there is a tiny amount of scope of a linear decision boundary, but it would not be very clean. Lets try a few others.

[17]:
```
plt.figure(figsize=(8, 5))
sns.scatterplot(data=df, x='Ripeness', y='Sweetness', hue='Quality')
plt.title("Dependence between ripeness and sweetness")
plt.show()
```



[18]:
```
plt.figure(figsize=(8, 5))
sns.scatterplot(data=df, x='Acidity', y='Sweetness', hue='Quality')
plt.title("Dependence between Acidity and sweetness")
plt.show()
```



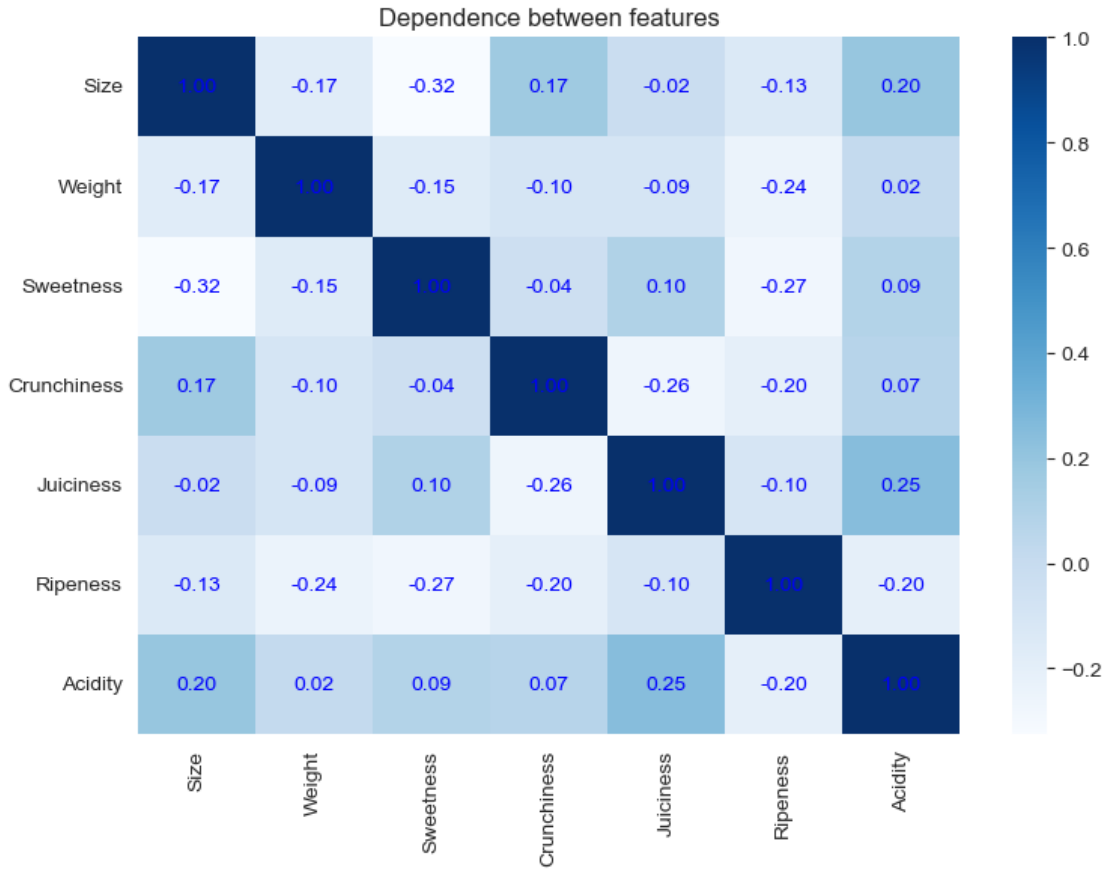
The three above pairwise plots all show that, at least for the set of 2D axes we selected, regarding linear seperability, the data is not cleanly linearly seperable for any of the 2D axes combinations shown above.

1.0.20 Correlation Matrix

One way of detecting if any of the features are very redundant with each other is by looking at the feature correlation matrix which is particularly easy to do when all features are real-valued and roughly bell shaped in histogram, which we have. We do this next.

[19]:
```
# We drop the 'Quality' column since it is our target variable and we're␣
 ↪interested in understanding the relationship between the features themselves.
df_feats = df.drop('Quality', axis=1)

# We plot the heatmap of the correlation matrix to understand the relationship␣
 ↪between the features.
plt.figure(figsize=(8, 6))
corr = df_feats.corr()
# Create the heatmap and show the correlations in each cell.
ax = sns.heatmap(corr, cmap='Blues')
plt.title("Dependence between features")
plt.tight_layout()
# Loop over data dimensions and create text annotations.
```

```
for i in range(corr.shape[0]):
    for j in range(corr.shape[1]):
        text = ax.text(j+0.5, i+0.5, "%.2f" % corr.iloc[i, j],
                       ha="center", va="center", color="b")

plt.show()
```

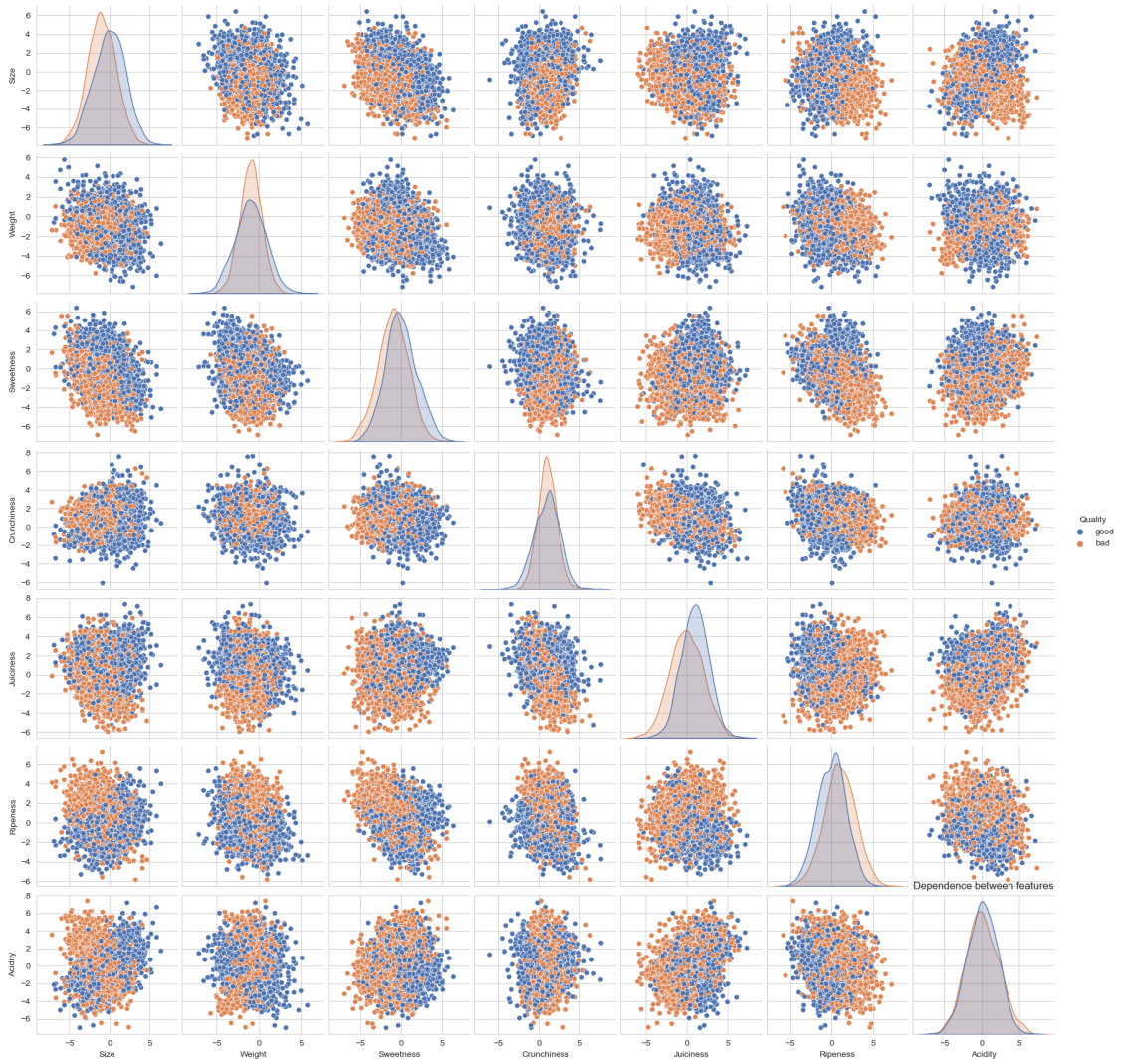


1.0.21 Many scatter plots

In the above we plotted only 3 scatter plots, but we can plot all m choose 2 of them at the same time as well as the feature histograms over the labels in one fell swoop. When you do not have too many features, this is a really useful plot to do to understand your data better, albeit realize that we're still only looking at all m choose 2 scatter plots, there still could be higher order relationships amongst size > 2 groups of features that these plots will not reveal.

[20]:
```
plt.figure()
sns.pairplot(df, hue='Quality')
plt.title("Dependence between features")
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



So the above plots tell three important but distinct properties about the data, namely that:

- (1) There are no redundant features in the data, as the correlation between features is low in general
- (2) Features such as size and weight, or sweetness and ripeness would intuituively (as per the physical world), be highly correlated. The data, however, shows otherwise
- (3) Data appears to be normalised, as it is centred around zero and does not seem very spread out.

1.0.22 Preparing data for prediction

What we next do is adjust the data to be ready to build a classifier. We'll first build several pre-built classifiers using sklearn, and then write code ourselves from scratch for a KNN classifier.

[21]:
```
# We split the data into features and target variable, X will contain the␣
 ↪features and y will contain the target variable.
# Note that X and y will still be DataFrames.
X = df.drop('Quality', axis=1)
y = df['Quality']
```

[22]:
```
# Next, we replace the 'good' and 'bad' values in the target variable with 1␣
 ↪and 0 respectively.
y = y.replace({'good': 1, 'bad': 0})
```

[23]:
```
# Lets double check that the values in the target variable have been replaced␣
 ↪by 0 and 1 values as expected.
y.head()
```

[23]:
```
A_id
0    1
1    1
2    0
3    1
4    1
Name: Quality, dtype: int64
```

[24]:
```
# Everyone in class should use the same training and test split, so we␣
 ↪explicitly set the random state to 97. Do not change this value!!!
# Also, this plot sets the test_size to 0.2 which means that 20% of the data␣
 ↪will be used for testing and 80% for training.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=97)
```

1.0.23 Prediction

Next we build three classifiers using built in sklearn code, namely we use pre-built random forest classifier, XGBoost classifier, and a decision tree classifier, lets see how they do.

[25]:
```
# What the `acc_score` function does is take an sklearn model and then fits the␣
 ↪model to the training data and then makes predictions on the test data.
# It then prints the accuracy score of the model on the test data.
def acc_score(model):
    model.fit(X_train, y_train)
    y_preds = model.predict(X_test)
    return print('ACC score:', accuracy_score(y_test, y_preds))
```

[26]:
```
# We start with a random forest classifier with 100 trees and a random state of␣
 ↪42.
forest_model_1 = RandomForestClassifier(n_estimators=100, random_state=42)
# We call the `acc_score` function with the `forest_model_1` model as an␣
 ↪argument.
```

```
acc_score(forest_model_1)
```

```
ACC score: 0.90125
```

[27]:
```
# We then try a random forest classifier with 1000 trees and a random state of␣
 ↪42.
# Does this boost the accuracy relative to the random forest classifier? Yes,␣
 ↪it does
forest_model_2 = RandomForestClassifier(n_estimators=1000, random_state=42)
# We call the `acc_score` function with the `forest_model_2` model as an␣
 ↪argument.
acc_score(forest_model_2)
```

```
ACC score: 0.905
```

[28]:
```
import xgboost as xgb
# Next, we try an XGBoost classifier with 100 trees and a random state of 42.
# Does this celebrated model family do better than the random forest? No, it␣
 ↪gives the same accuracy as the random forest model
xgb_model_1 = xgb.XGBClassifier(objective='binary:logistic', n_estimators=100,␣
 ↪seed=42)
# We call the `acc_score` function with the `xgb_model_1` model as an argument.
acc_score(xgb_model_1)
```

```
ACC score: 0.90125
```

[29]:
```
# Next we check if the XGBoost classifier with 100 trees is statistically␣
 ↪significantly better than the random forest classifier with 1000 trees.
# We can do this by using a t-test to compare the two models.
from scipy.stats import ttest_rel
y_preds_xgb = xgb_model_1.predict(X_test)
y_preds_forest = forest_model_2.predict(X_test)
ttest_rel(y_preds_xgb, y_preds_forest)
# The output is statistic=-1.480693188626695, pvalue=0.13908255678067352,␣
 ↪df=799. So there is no evidence that either classifier is statistically␣
 ↪significantly better than the other
# Thus, do we reject or do we fail to reject the null hypothesis that the two␣
 ↪models are statistically significantly different? Clearly, the pvalue is␣
 ↪greater than 0.05, so there is no evidence to reject the null hypothesis.␣
 ↪Hence, we fail to reject the null hypothesis
```

[29]:
```
TtestResult(statistic=-1.480693188626695, pvalue=0.13908255678067352, df=799)
```

[30]:
```
# Next, we try an XGBoost classifier with 1000 trees and a random state of 42.␣
 ↪Will this boost the accuracy more than the above classifiers?
xgb_model_2 = xgb.XGBClassifier(objective='binary:logistic', n_estimators=1000,␣
 ↪seed=42)
```

```
# We call the `acc_score` function with the `xgb_model_2` model as an argument.
acc_score(xgb_model_2)
```

```
ACC score: 0.8975
```

[31]:
```
# Last but not least, we try a decision tree classifier with a random state of␣
 ↪42.
decision_model = DecisionTreeClassifier(random_state=42)
# We call the `acc_score` function with the `decision_model` model as an␣
 ↪argument.
acc_score(decision_model)
```

```
ACC score: 0.80625
```

1.0.24 Conclusion on pre-built classifiers.

We should see some variability of the performance of each of the classifiers.

You are to record the accuracies of the sklearn models in the dataset and compare them with your own KNN classifier below.

Looking at the above results, we conclude that: - Random forest classifier seems to improve in performance as we increase the number of trees, while the same is not the case for the GBDT - Standard decision trees show worse performance than random forest and GBDT

1.0.25 On to building our own KNN classifier.

Having established the above baselines to compare against (which is *always* a very good idea to do), we next build our own KNN decision tree code and see if we can match or beat the above. We will use the variable `kvalue` to be the number of nearest neighbors that our KNN classifier uses.

[32]:
```
from collections import Counter
from tqdm.autonotebook import tqdm

# This function will calculate the Euclidean distance between two rows of data.
def euclidean_distance(row1, row2):
    distance = np.linalg.norm(row1 - row2)
    return distance
```

[33]:
```
# KNN Algorithm. This function takes in the training data, the test data and␣
 ↪the value of k.
# It runs the KNN algorithm using the kvalue `k` and returns the predicted␣
 ↪values for the given test data
# as a numpy array.


# Here you are to provide your own code for a KNN classifier. You may if you␣
 ↪like use the `euclidean_distance` function you provided above to help you.
def knn(X_train, y_train, X_test, kvalue):
```

```
    y_pred = []
    for x in X_test :
        distances = [euclidean_distance(x, x_train) for x_train in X_train]
        k_indices = np.argsort(distances)[:kvalue]
        k_nearest_labels = [y_train[i] for i in k_indices]
        most_common = np.argmax(np.bincount(k_nearest_labels))
        y_pred.append(most_common)
    return np.array(y_pred)
```

[34]:
```
# This function computes the accuracy of a model by comparing the predicted␣
 ↪values with the actual values, both of which
# are given in the arguments. The function returns the accuracy as a float.
def accuracy(y_true, y_pred):
    accuracy = np.sum(y_true == y_pred) / len(y_true)
    return accuracy
```

[35]:
```
# This sets up testing for different values of kvalue
k_values = range(1, 200, 1)
```

[36]:
```
# We next ensure that the data is in NumPy array format since the code above␣
 ↪uses numpy matrices and arrays, not pandas DataFrames.
X_train_np = X_train.to_numpy()
X_test_np = X_test.to_numpy()
y_train_np = y_train.to_numpy().ravel()  # ravel in case y_train is a column␣
 ↪vector
y_test_np = y_test.to_numpy().ravel()    # ravel in case y_test is a column␣
 ↪vector
```

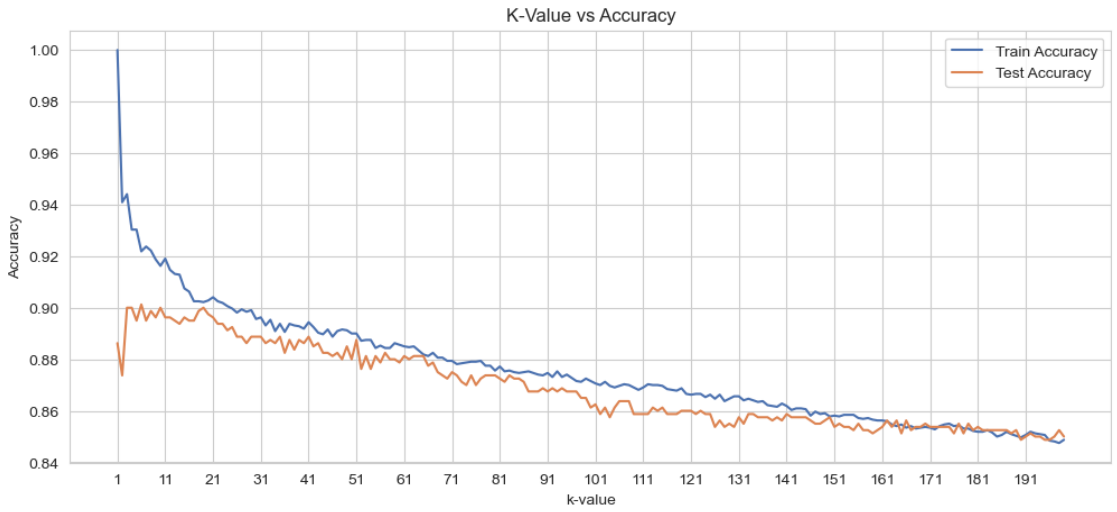
[37]:
```
# Reset the accuracies arrays to empty lists
train_accuracies = []
test_accuracies = []
# We loop over the different values of k and calculate the accuracy for each␣
 ↪value of k.
for kvalue in tqdm(k_values, position=0, desc='k_values Progress'):
    # Predict labels for train set and test set
    y_pred_train = knn(X_train_np, y_train_np, X_train_np, kvalue)
    y_pred_test = knn(X_train_np, y_train_np, X_test_np, kvalue)

    # Calculate accuracy
    train_accuracies.append(accuracy(y_train_np, y_pred_train))
    test_accuracies.append(accuracy(y_test_np, y_pred_test))
```

```
k_values Progress:   0%|          | 0/199 [00:00<?, ?it/s]
```

[38]:
```
# We plot the results of our KNN classifier to see how the accuracy changes␣
 ↪with the value of kvalue on both the training and test data,
# seeing where the underfitting and overfitting regions are.
```

```
# Plotting
plt.figure(figsize=(12, 5))
plt.plot(k_values, train_accuracies, label='Train Accuracy')
plt.plot(k_values, test_accuracies, label='Test Accuracy')
plt.legend()
plt.title('K-Value vs Accuracy')
plt.xlabel('k-value')
plt.ylabel('Accuracy')
#plt.xticks(list(k_values))  # Ensure that x-axis ticks match k values
tick_interval = 10
plt.xticks(range(1, len(k_values) + 1, tick_interval))
print('Best k-value:', k_values[np.argmax(test_accuracies)])
print('Max accuracy:', max(test_accuracies))
plt.show()
```

```
Best k-value: 6
Max accuracy: 0.90125
```



1.0.26 Questions

These qre questions in the pdf file they are answered here.

Q1: What kvalue had the best training accuracy?

A1: k = 1

Q2: What kvalue had the best testing accuracy?

A2: k = 6

Q3: What range of kvalues lead to underfitting?

A3: k > = 160, as per the graph

Q4: What range of kvalues lead to overfitting?

A4: k in [1,5]

Q5: Carefully explain and justify your answers by referring to specific regions of the plot and explain your knowledge of overfitting, underfitting, and fitting.

A5: Underfitting is the region where a model has high bias and low variance, and where test accuracy would outperform training accuracy. Overfitting the region of this plot where a model has high variance and low bias, and where the training accuracy would outperform the test accuracy, as the model would be too specific to the training set. Finally, fitting is the region where the model has good bias and variance, such that the testing accuracy is typically at its highest, but is generally still slightly lower than training accuracy. Therefore, based on the above, in the graph, k=6 is the ideal fit of the model, k<6 is the region for overfitting, and underfitting occurs for k > =160.