CA3

Imports

In [2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, IsolationForest
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
```

Reading data

In [17]:
```
df = pd.read_csv("./assets/train.csv", index_col = 0)

df.info()

# Checking for missing values
print(f"\nMissing values inn training data: {df.isnull().sum().sum()}")
```

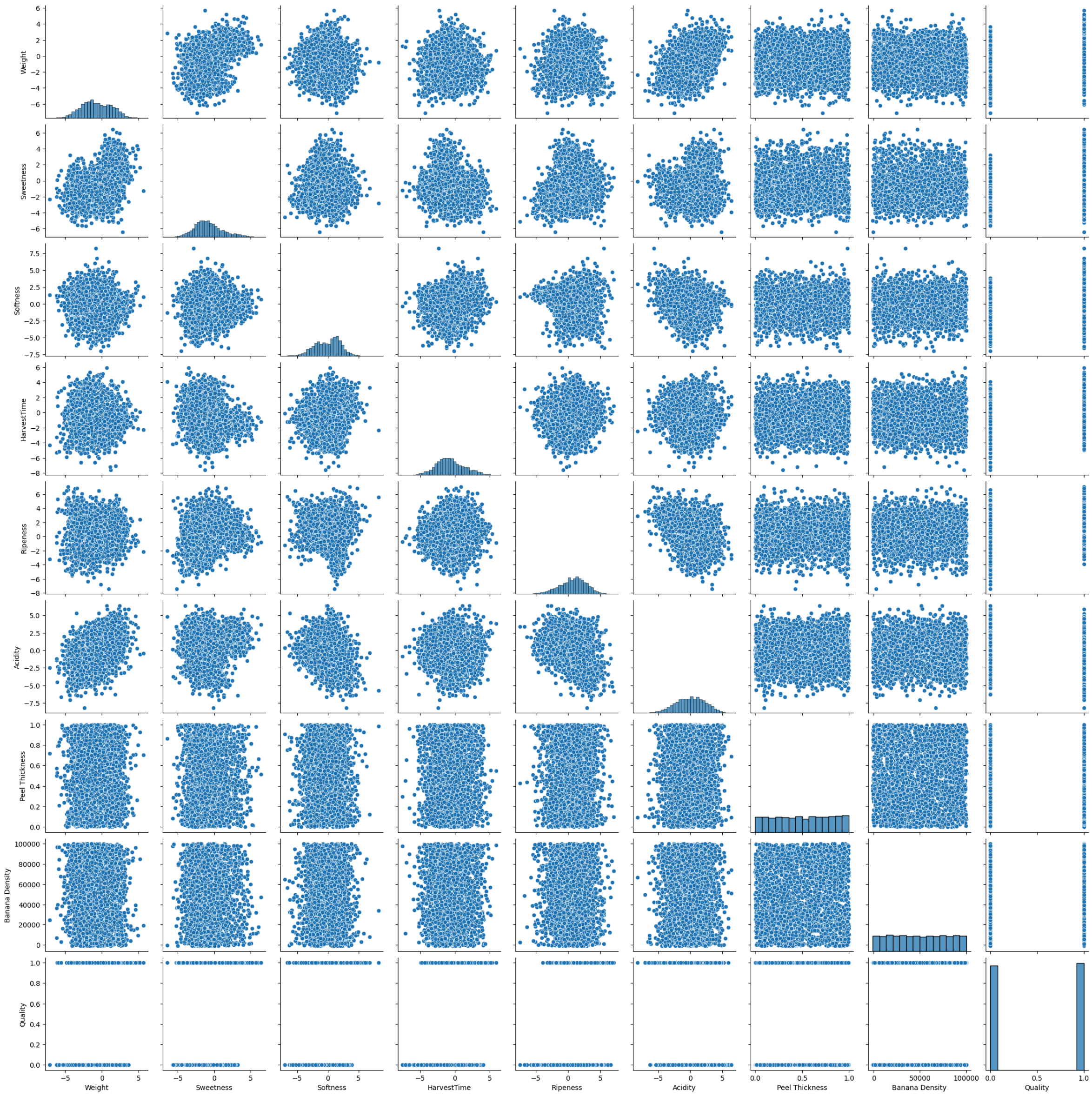
```
<class 'pandas.core.frame.DataFrame'>
Index: 2800 entries, -1.8257343 to -1.6260979
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Weight          2800 non-null   float64
 1   Sweetness       2800 non-null   float64
 2   Softness        2800 non-null   float64
 3   HarvestTime     2800 non-null   float64
 4   Ripeness        2800 non-null   float64
 5   Acidity         2800 non-null   float64
 6   Peel Thickness  2800 non-null   float64
 7   Banana Density  2800 non-null   float64
 8   Quality         2800 non-null   int64  
dtypes: float64(8), int64(1)
memory usage: 218.8 KB

Missing values inn training data: 0
```

Data exploration and visualisation

In [4]:
```
print(df.describe()) # Gives a table of the dataset with statistical components

sns.pairplot(df) # Gives a scatter plot for every pair & a histogram for the the
```

```
            Weight    Sweetness     Softness  HarvestTime     Ripeness  \
count  2800.000000  2800.000000  2800.000000  2800.000000  2800.000000   
mean     -0.751050    -0.751005    -0.019557    -0.700683     0.771011   
std       2.006590     1.955109     2.076865     2.029916     2.098275   
min      -7.103426    -6.434022    -6.959320    -7.570008    -7.423155   
25%      -2.238843    -2.104742    -1.593816    -2.112747    -0.572589   
50%      -0.882387    -0.997902     0.220174    -0.856858     0.930927   
75%       0.853566     0.334989     1.542899     0.628895     2.229410   
max       5.679692     6.438196     8.241555     5.942060     7.077372   

          Acidity  Peel Thickness  Banana Density      Quality  
count  2800.000000     2800.000000     2800.000000  2800.000000  
mean     -0.000989        0.506758    49397.491271     0.506429  
std       2.286725        0.291936    29327.077623     0.500048  
min      -8.226977        0.000086     -980.343999     0.000000  
25%      -1.608385        0.257860    24025.427350     0.000000  
50%       0.073963        0.506282    49303.534616     1.000000  
75%       1.662417        0.761016    75066.598785     1.000000  
max       6.395850        0.999430    99982.761410     1.000000  
```

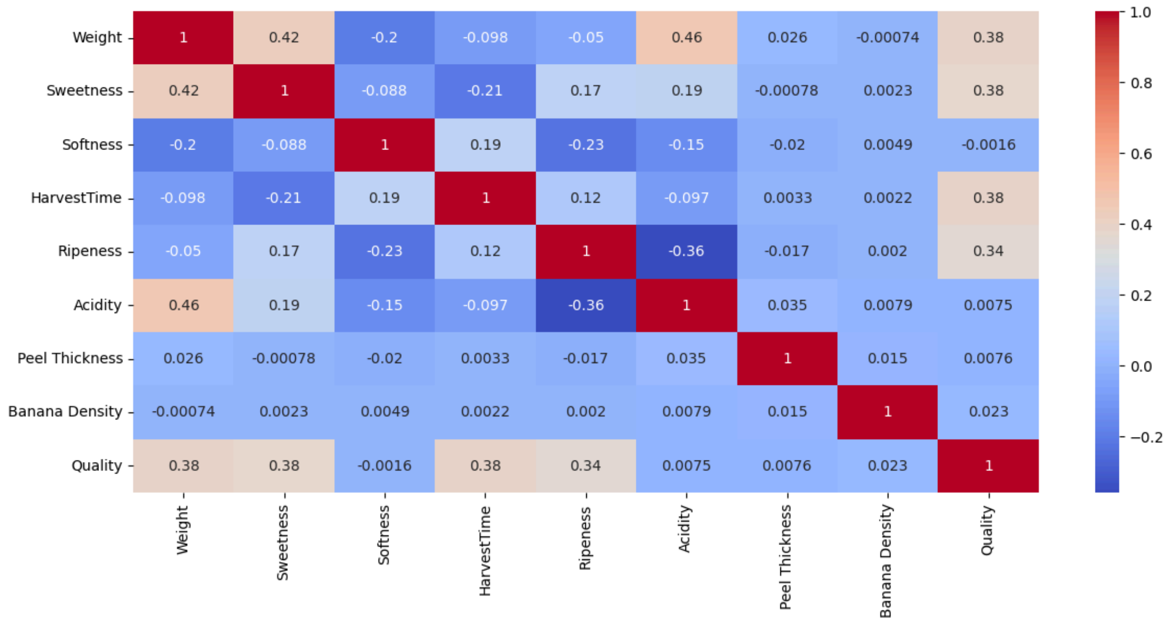
Out[4]: `<seaborn.axisgrid.PairGrid at 0x2d288a22610>`



The pairs of different features does not look linear seperable, they have a complex relation.
Therefore, we think models like logistic regression will have low accuracy for theese data.

In [5]:
```
""" Checking correlation with every feature """
fig, axes = plt.subplots(1, 1, figsize=(14, 6)) # To make the values more readab
# annot = true to get value, coolwarm cmap to make sense with correlation
sns.heatmap(df.corr(), annot = True, cmap = "coolwarm")
```

Out[5]: `<Axes: >`



As we can see: Acidity and Weight, and Weight and Sweetness has the highest positive correlation.
While the Acidity and Ripeness, and Softness and Ripeness has the highest negative correlation.

To conclude:

The pairs of different features have complex relations to each other. The data is not linearly seperable and
therefore, we think models like logistic regression will have low accuracy for theese data.
Acidity and Weight, Weight and Sweetness has the highest positive correlation.
While the Acidity and Ripeness, and Softness and Ripeness has the highest negative correlation.
With this informasjotion we know which features we can focus more on.
We also found out that the banana density values are too high. We want to scale and standardize
the values of the dataset to fit the models better.
We will probably remove the coloumns banana density and peel thickness because of the low correlation
to the other features.

Data Preprocessing & Feature Engineering

We found out in the exploration and visualization part that there were not any missing values.
However, we will still drop outliers if we find any.

Feature Selection: We saw that Banana Density and Peel Thickness has low correlation with other features.

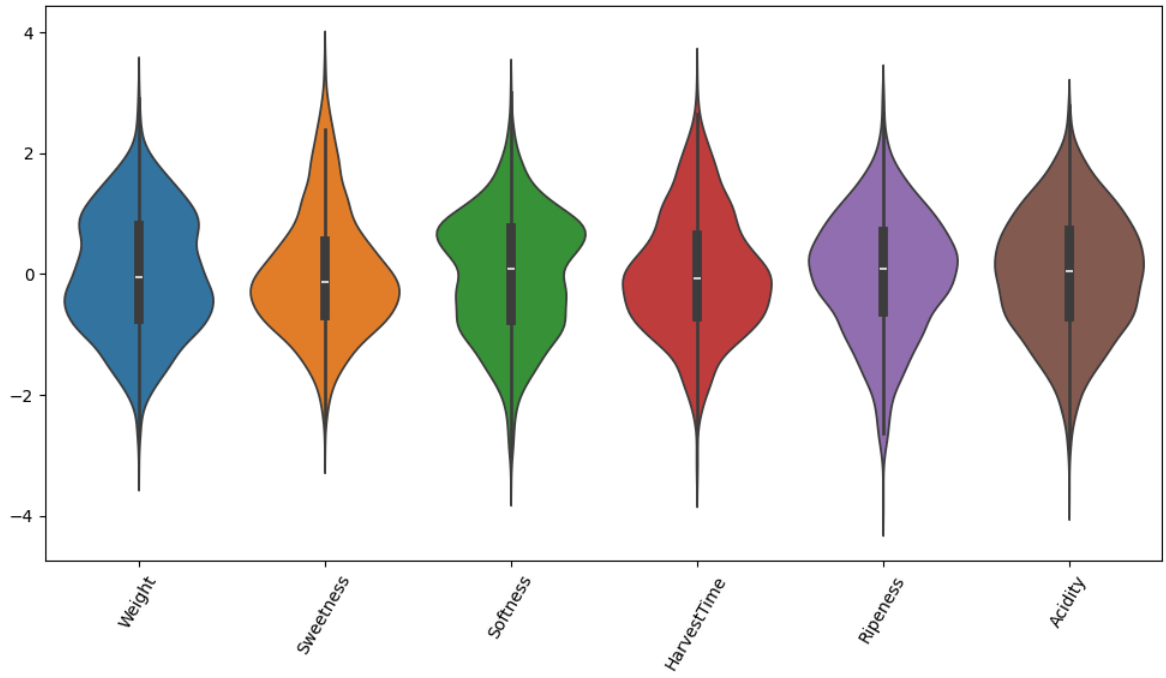
Therefore we will drop those features

In [19]:
```
# Feature Selection, we are dropping the features when we're dropping coloumns i
useless_features = ["Banana Density", "Peel Thickness"]

# Including outliers, we found out that our model gets worse by removing outlier

df_clean = df.copy()
X = df_clean.drop(columns = ["Quality"] + useless_features)
y = df_clean["Quality"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, rand
```

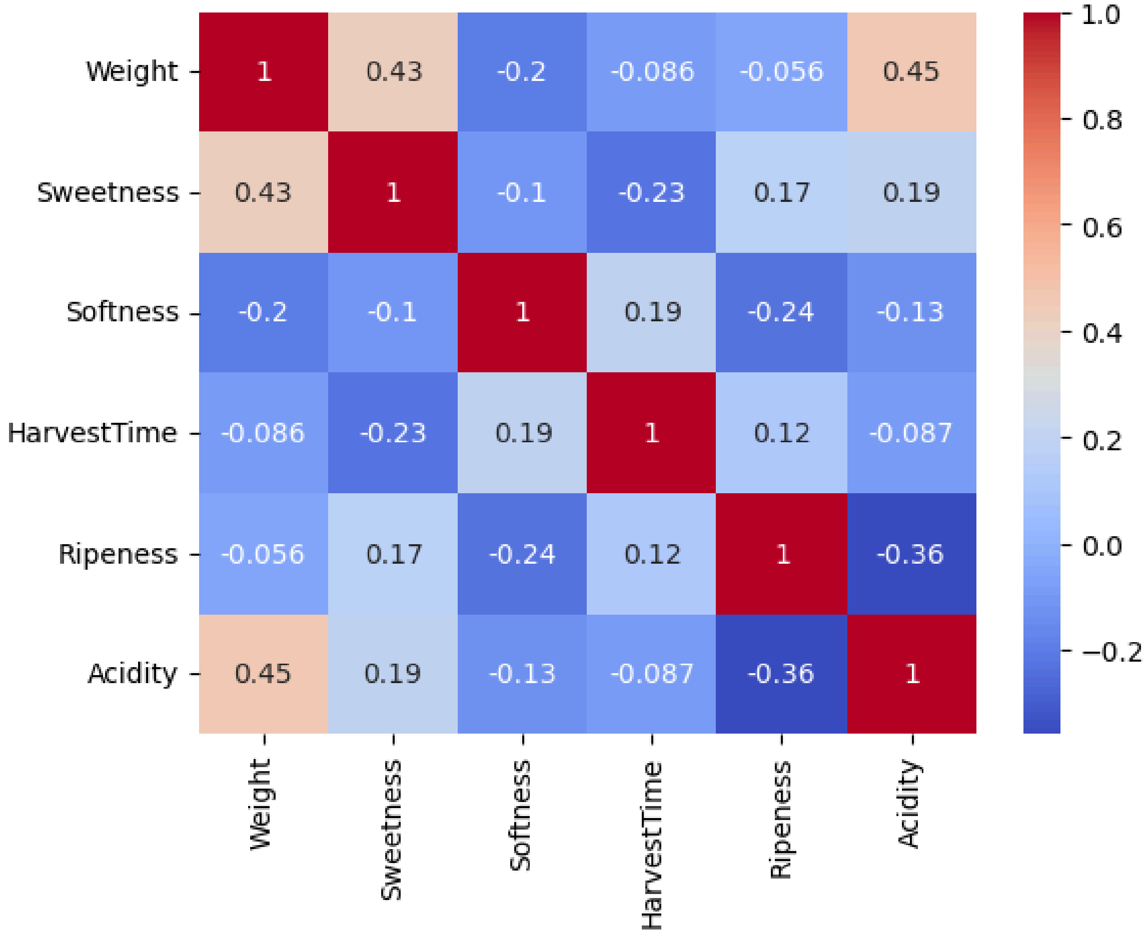
Data preprocessing and visualisation

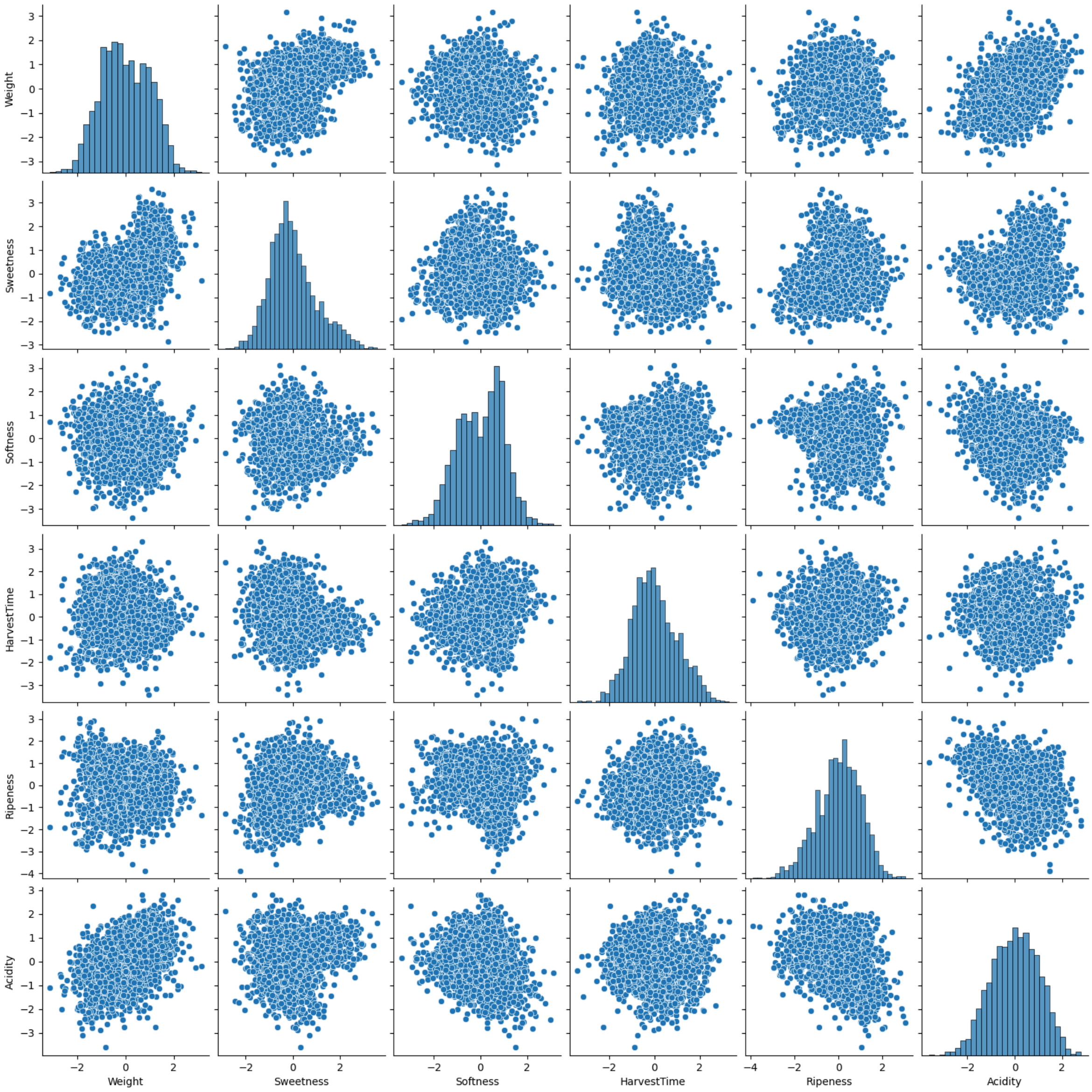
In [20]:
```
# Standardizing
scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)  # Using transform to prevent data leak

print("Means:", X_train_scaled.mean(axis = 0))
print("\nStandard deviations:", X_train_scaled.std(axis = 0))

# Final Visualization
df_train_scaled = pd.DataFrame(X_train_scaled, columns = df_clean.drop(columns =

# Violin plots to visualize distribution
plt.figure(figsize = (12, 6))
sns.violinplot(data = df_train_scaled)
plt.xticks(rotation=60)
plt.show()

# Heatmap for visualizing correlation after data cleaning.
sns.heatmap(df_train_scaled.corr(), annot = True, cmap = "coolwarm")

# Pairplot for visualizing relations again.
sns.pairplot(df_train_scaled)
```

```
Means: [-1.94553368e-17  1.77635684e-17 -2.62224105e-17  1.69176842e-17
 -2.07241631e-17  0.00000000e+00]

Standard deviations: [1. 1. 1. 1. 1. 1.]
```




```
Out[20]: <seaborn.axisgrid.PairGrid at 0x2d2a5bc7610>
```





It looks pretty much the same. However the correlation has just changed a bit.
For example weight-adicity correlation changed with -0.01.

Modelling

Finding good parameters

In [9]:
```
# Logistic Regression Classifier
param_grid_LR = {
    'C': [0.01, 0.1, 1, 10, 100],
}
grid_search_LR = GridSearchCV(LogisticRegression(), param_grid_LR, cv=5, n_jobs=
grid_search_LR.fit(X_train_scaled, y_train)

print(grid_search_LR.best_params_)
```

```
{'C': 1}
```

In [10]:
```
# SVC
param_grid_SVC = {
    'C': [0.01, 0.1, 1, 10, 11],
    'kernel': ['linear', 'rbf', 'poly'],
    'gamma': ['scale', 'auto'],
```

```
}
grid_search_SVC = GridSearchCV(SVC(), param_grid_SVC, cv=5, n_jobs=-1)
grid_search_SVC.fit(X_train_scaled, y_train)

print(grid_search_SVC.best_params_)
```

```
{'C': 11, 'gamma': 'scale', 'kernel': 'rbf'}
```

In [11]:
```
# DecisionTree
param_grid_DT = {
    'max_depth': [3, 10, 20, 30],
    'min_samples_split': [2, 5, 7, 8, 9, 10],
    'criterion': ['gini', 'entropy']
}
grid_search_DT = GridSearchCV(DecisionTreeClassifier(), param_grid_DT, cv=5, n_j
grid_search_DT.fit(X_train_scaled, y_train)

print(grid_search_DT.best_params_)
```

```
{'criterion': 'entropy', 'max_depth': 10, 'min_samples_split': 5}
```

In [12]:
```
# Random Forest
param_grid_RF = {
    'n_estimators': [100, 125, 200],
    'max_depth': [15, 20, 21, 25],
    'criterion': ['gini', 'entropy']
}
grid_search_RF = GridSearchCV(RandomForestClassifier(), param_grid_RF, cv=5, n_j
grid_search_RF.fit(X_train_scaled, y_train)

print(grid_search_RF.best_params_)
```

```
{'criterion': 'gini', 'max_depth': 15, 'n_estimators': 200}
```

In [13]:
```
# KNN
param_grid_KNN = {
    'n_neighbors': [3, 5, 7, 10, 15],
    'p': [1, 2, 3, 4, 5, 6, 7]
}
grid_search_KNN = GridSearchCV(KNeighborsClassifier(), param_grid_KNN, cv=5, n_j
grid_search_KNN.fit(X_train_scaled, y_train)

print(grid_search_KNN.best_params_)
```

```
{'n_neighbors': 5, 'p': 5}
```

Finding the best model

In [14]:
```
dataset_sizes = np.arange(10, 2800, 50)

X_train_scaled = np.array(X_train_scaled)
y_train = np.array(y_train)

models = []

# Playing around with the variables to find something good
classifiers = {
    "LogisticRegression": LogisticRegression(C = 1),
    "SVC": SVC(C = 11, gamma = "scale", kernel = "rbf"),
    "DecisionTree": DecisionTreeClassifier(criterion = "entropy", max_depth = 10
    "RandomForest": RandomForestClassifier(criterion = "gini", max_depth = 20, n
```

```python
    "KNN": KNeighborsClassifier(n_neighbors = 5, p = 5),
}

keys = classifiers.keys()

for clf_name in keys:
    clf = classifiers[clf_name]
    clf_index = list(keys).index(clf_name)

    for size_index, size in enumerate(dataset_sizes):
        X_train_subset, y_train_subset = X_train_scaled[:size], y_train[:size]

        clf.fit(X_train_subset, y_train_subset)

        y_pred = clf.predict(X_test_scaled)
        accuracy = accuracy_score(y_test, y_pred)

        models.append((clf, accuracy, size))

        print(f"{clf_name} | Size: {size} | Accuracy: {accuracy:.4f}")
```

```
LogisticRegression | Size: 10 | Accuracy: 0.7857
LogisticRegression | Size: 60 | Accuracy: 0.8600
LogisticRegression | Size: 110 | Accuracy: 0.8671
LogisticRegression | Size: 160 | Accuracy: 0.8643
LogisticRegression | Size: 210 | Accuracy: 0.8614
LogisticRegression | Size: 260 | Accuracy: 0.8600
LogisticRegression | Size: 310 | Accuracy: 0.8643
LogisticRegression | Size: 360 | Accuracy: 0.8657
LogisticRegression | Size: 410 | Accuracy: 0.8600
LogisticRegression | Size: 460 | Accuracy: 0.8600
LogisticRegression | Size: 510 | Accuracy: 0.8600
LogisticRegression | Size: 560 | Accuracy: 0.8586
LogisticRegression | Size: 610 | Accuracy: 0.8614
LogisticRegression | Size: 660 | Accuracy: 0.8657
LogisticRegression | Size: 710 | Accuracy: 0.8629
LogisticRegression | Size: 760 | Accuracy: 0.8657
LogisticRegression | Size: 810 | Accuracy: 0.8657
LogisticRegression | Size: 860 | Accuracy: 0.8700
LogisticRegression | Size: 910 | Accuracy: 0.8700
LogisticRegression | Size: 960 | Accuracy: 0.8671
LogisticRegression | Size: 1010 | Accuracy: 0.8629
LogisticRegression | Size: 1060 | Accuracy: 0.8614
LogisticRegression | Size: 1110 | Accuracy: 0.8629
LogisticRegression | Size: 1160 | Accuracy: 0.8657
LogisticRegression | Size: 1210 | Accuracy: 0.8657
LogisticRegression | Size: 1260 | Accuracy: 0.8671
LogisticRegression | Size: 1310 | Accuracy: 0.8686
LogisticRegression | Size: 1360 | Accuracy: 0.8671
LogisticRegression | Size: 1410 | Accuracy: 0.8671
LogisticRegression | Size: 1460 | Accuracy: 0.8686
LogisticRegression | Size: 1510 | Accuracy: 0.8686
LogisticRegression | Size: 1560 | Accuracy: 0.8671
LogisticRegression | Size: 1610 | Accuracy: 0.8686
LogisticRegression | Size: 1660 | Accuracy: 0.8700
LogisticRegression | Size: 1710 | Accuracy: 0.8700
LogisticRegression | Size: 1760 | Accuracy: 0.8700
LogisticRegression | Size: 1810 | Accuracy: 0.8729
LogisticRegression | Size: 1860 | Accuracy: 0.8729
LogisticRegression | Size: 1910 | Accuracy: 0.8700
LogisticRegression | Size: 1960 | Accuracy: 0.8729
LogisticRegression | Size: 2010 | Accuracy: 0.8714
LogisticRegression | Size: 2060 | Accuracy: 0.8729
LogisticRegression | Size: 2110 | Accuracy: 0.8714
LogisticRegression | Size: 2160 | Accuracy: 0.8714
LogisticRegression | Size: 2210 | Accuracy: 0.8714
LogisticRegression | Size: 2260 | Accuracy: 0.8714
LogisticRegression | Size: 2310 | Accuracy: 0.8714
LogisticRegression | Size: 2360 | Accuracy: 0.8714
LogisticRegression | Size: 2410 | Accuracy: 0.8714
LogisticRegression | Size: 2460 | Accuracy: 0.8714
LogisticRegression | Size: 2510 | Accuracy: 0.8714
LogisticRegression | Size: 2560 | Accuracy: 0.8714
LogisticRegression | Size: 2610 | Accuracy: 0.8714
LogisticRegression | Size: 2660 | Accuracy: 0.8714
LogisticRegression | Size: 2710 | Accuracy: 0.8714
LogisticRegression | Size: 2760 | Accuracy: 0.8714
SVC | Size: 10 | Accuracy: 0.9000
SVC | Size: 60 | Accuracy: 0.9171
SVC | Size: 110 | Accuracy: 0.9200
SVC | Size: 160 | Accuracy: 0.9357
```

```
SVC | Size: 210 | Accuracy: 0.9529
SVC | Size: 260 | Accuracy: 0.9543
SVC | Size: 310 | Accuracy: 0.9500
SVC | Size: 360 | Accuracy: 0.9514
SVC | Size: 410 | Accuracy: 0.9571
SVC | Size: 460 | Accuracy: 0.9586
SVC | Size: 510 | Accuracy: 0.9657
SVC | Size: 560 | Accuracy: 0.9629
SVC | Size: 610 | Accuracy: 0.9671
SVC | Size: 660 | Accuracy: 0.9657
SVC | Size: 710 | Accuracy: 0.9643
SVC | Size: 760 | Accuracy: 0.9686
SVC | Size: 810 | Accuracy: 0.9700
SVC | Size: 860 | Accuracy: 0.9700
SVC | Size: 910 | Accuracy: 0.9671
SVC | Size: 960 | Accuracy: 0.9657
SVC | Size: 1010 | Accuracy: 0.9643
SVC | Size: 1060 | Accuracy: 0.9671
SVC | Size: 1110 | Accuracy: 0.9714
SVC | Size: 1160 | Accuracy: 0.9686
SVC | Size: 1210 | Accuracy: 0.9714
SVC | Size: 1260 | Accuracy: 0.9714
SVC | Size: 1310 | Accuracy: 0.9714
SVC | Size: 1360 | Accuracy: 0.9729
SVC | Size: 1410 | Accuracy: 0.9729
SVC | Size: 1460 | Accuracy: 0.9729
SVC | Size: 1510 | Accuracy: 0.9729
SVC | Size: 1560 | Accuracy: 0.9714
SVC | Size: 1610 | Accuracy: 0.9729
SVC | Size: 1660 | Accuracy: 0.9729
SVC | Size: 1710 | Accuracy: 0.9700
SVC | Size: 1760 | Accuracy: 0.9700
SVC | Size: 1810 | Accuracy: 0.9700
SVC | Size: 1860 | Accuracy: 0.9714
SVC | Size: 1910 | Accuracy: 0.9714
SVC | Size: 1960 | Accuracy: 0.9700
SVC | Size: 2010 | Accuracy: 0.9714
SVC | Size: 2060 | Accuracy: 0.9714
SVC | Size: 2110 | Accuracy: 0.9700
SVC | Size: 2160 | Accuracy: 0.9700
SVC | Size: 2210 | Accuracy: 0.9700
SVC | Size: 2260 | Accuracy: 0.9700
SVC | Size: 2310 | Accuracy: 0.9700
SVC | Size: 2360 | Accuracy: 0.9700
SVC | Size: 2410 | Accuracy: 0.9700
SVC | Size: 2460 | Accuracy: 0.9700
SVC | Size: 2510 | Accuracy: 0.9700
SVC | Size: 2560 | Accuracy: 0.9700
SVC | Size: 2610 | Accuracy: 0.9700
SVC | Size: 2660 | Accuracy: 0.9700
SVC | Size: 2710 | Accuracy: 0.9700
SVC | Size: 2760 | Accuracy: 0.9700
DecisionTree | Size: 10 | Accuracy: 0.5371
DecisionTree | Size: 60 | Accuracy: 0.7886
DecisionTree | Size: 110 | Accuracy: 0.7643
DecisionTree | Size: 160 | Accuracy: 0.8114
DecisionTree | Size: 210 | Accuracy: 0.8414
DecisionTree | Size: 260 | Accuracy: 0.8529
DecisionTree | Size: 310 | Accuracy: 0.8657
DecisionTree | Size: 360 | Accuracy: 0.8900
```

```
DecisionTree | Size: 410 | Accuracy: 0.8514
DecisionTree | Size: 460 | Accuracy: 0.8714
DecisionTree | Size: 510 | Accuracy: 0.8557
DecisionTree | Size: 560 | Accuracy: 0.8971
DecisionTree | Size: 610 | Accuracy: 0.8986
DecisionTree | Size: 660 | Accuracy: 0.9029
DecisionTree | Size: 710 | Accuracy: 0.9000
DecisionTree | Size: 760 | Accuracy: 0.9129
DecisionTree | Size: 810 | Accuracy: 0.9057
DecisionTree | Size: 860 | Accuracy: 0.9114
DecisionTree | Size: 910 | Accuracy: 0.9129
DecisionTree | Size: 960 | Accuracy: 0.9271
DecisionTree | Size: 1010 | Accuracy: 0.8971
DecisionTree | Size: 1060 | Accuracy: 0.9114
DecisionTree | Size: 1110 | Accuracy: 0.9057
DecisionTree | Size: 1160 | Accuracy: 0.9129
DecisionTree | Size: 1210 | Accuracy: 0.9100
DecisionTree | Size: 1260 | Accuracy: 0.9143
DecisionTree | Size: 1310 | Accuracy: 0.9000
DecisionTree | Size: 1360 | Accuracy: 0.9143
DecisionTree | Size: 1410 | Accuracy: 0.9086
DecisionTree | Size: 1460 | Accuracy: 0.9129
DecisionTree | Size: 1510 | Accuracy: 0.9129
DecisionTree | Size: 1560 | Accuracy: 0.9100
DecisionTree | Size: 1610 | Accuracy: 0.9200
DecisionTree | Size: 1660 | Accuracy: 0.9086
DecisionTree | Size: 1710 | Accuracy: 0.9129
DecisionTree | Size: 1760 | Accuracy: 0.9171
DecisionTree | Size: 1810 | Accuracy: 0.9129
DecisionTree | Size: 1860 | Accuracy: 0.9143
DecisionTree | Size: 1910 | Accuracy: 0.9286
DecisionTree | Size: 1960 | Accuracy: 0.9314
DecisionTree | Size: 2010 | Accuracy: 0.9229
DecisionTree | Size: 2060 | Accuracy: 0.9300
DecisionTree | Size: 2110 | Accuracy: 0.9229
DecisionTree | Size: 2160 | Accuracy: 0.9229
DecisionTree | Size: 2210 | Accuracy: 0.9200
DecisionTree | Size: 2260 | Accuracy: 0.9200
DecisionTree | Size: 2310 | Accuracy: 0.9214
DecisionTree | Size: 2360 | Accuracy: 0.9243
DecisionTree | Size: 2410 | Accuracy: 0.9229
DecisionTree | Size: 2460 | Accuracy: 0.9186
DecisionTree | Size: 2510 | Accuracy: 0.9271
DecisionTree | Size: 2560 | Accuracy: 0.9243
DecisionTree | Size: 2610 | Accuracy: 0.9243
DecisionTree | Size: 2660 | Accuracy: 0.9214
DecisionTree | Size: 2710 | Accuracy: 0.9257
DecisionTree | Size: 2760 | Accuracy: 0.9200
RandomForest | Size: 10 | Accuracy: 0.7586
RandomForest | Size: 60 | Accuracy: 0.8743
RandomForest | Size: 110 | Accuracy: 0.8671
RandomForest | Size: 160 | Accuracy: 0.8986
RandomForest | Size: 210 | Accuracy: 0.9100
RandomForest | Size: 260 | Accuracy: 0.9200
RandomForest | Size: 310 | Accuracy: 0.9314
RandomForest | Size: 360 | Accuracy: 0.9443
RandomForest | Size: 410 | Accuracy: 0.9386
RandomForest | Size: 460 | Accuracy: 0.9471
RandomForest | Size: 510 | Accuracy: 0.9457
RandomForest | Size: 560 | Accuracy: 0.9471
```

```
RandomForest | Size: 610 | Accuracy: 0.9586
RandomForest | Size: 660 | Accuracy: 0.9514
RandomForest | Size: 710 | Accuracy: 0.9486
RandomForest | Size: 760 | Accuracy: 0.9571
RandomForest | Size: 810 | Accuracy: 0.9529
RandomForest | Size: 860 | Accuracy: 0.9543
RandomForest | Size: 910 | Accuracy: 0.9614
RandomForest | Size: 960 | Accuracy: 0.9557
RandomForest | Size: 1010 | Accuracy: 0.9557
RandomForest | Size: 1060 | Accuracy: 0.9557
RandomForest | Size: 1110 | Accuracy: 0.9543
RandomForest | Size: 1160 | Accuracy: 0.9586
RandomForest | Size: 1210 | Accuracy: 0.9557
RandomForest | Size: 1260 | Accuracy: 0.9557
RandomForest | Size: 1310 | Accuracy: 0.9543
RandomForest | Size: 1360 | Accuracy: 0.9557
RandomForest | Size: 1410 | Accuracy: 0.9543
RandomForest | Size: 1460 | Accuracy: 0.9529
RandomForest | Size: 1510 | Accuracy: 0.9571
RandomForest | Size: 1560 | Accuracy: 0.9529
RandomForest | Size: 1610 | Accuracy: 0.9571
RandomForest | Size: 1660 | Accuracy: 0.9543
RandomForest | Size: 1710 | Accuracy: 0.9529
RandomForest | Size: 1760 | Accuracy: 0.9571
RandomForest | Size: 1810 | Accuracy: 0.9571
RandomForest | Size: 1860 | Accuracy: 0.9600
RandomForest | Size: 1910 | Accuracy: 0.9557
RandomForest | Size: 1960 | Accuracy: 0.9543
RandomForest | Size: 2010 | Accuracy: 0.9586
RandomForest | Size: 2060 | Accuracy: 0.9543
RandomForest | Size: 2110 | Accuracy: 0.9557
RandomForest | Size: 2160 | Accuracy: 0.9586
RandomForest | Size: 2210 | Accuracy: 0.9571
RandomForest | Size: 2260 | Accuracy: 0.9571
RandomForest | Size: 2310 | Accuracy: 0.9557
RandomForest | Size: 2360 | Accuracy: 0.9586
RandomForest | Size: 2410 | Accuracy: 0.9557
RandomForest | Size: 2460 | Accuracy: 0.9557
RandomForest | Size: 2510 | Accuracy: 0.9586
RandomForest | Size: 2560 | Accuracy: 0.9600
RandomForest | Size: 2610 | Accuracy: 0.9600
RandomForest | Size: 2660 | Accuracy: 0.9571
RandomForest | Size: 2710 | Accuracy: 0.9586
RandomForest | Size: 2760 | Accuracy: 0.9600
KNN | Size: 10 | Accuracy: 0.6429
KNN | Size: 60 | Accuracy: 0.9243
KNN | Size: 110 | Accuracy: 0.9386
KNN | Size: 160 | Accuracy: 0.9486
KNN | Size: 210 | Accuracy: 0.9514
KNN | Size: 260 | Accuracy: 0.9557
KNN | Size: 310 | Accuracy: 0.9557
KNN | Size: 360 | Accuracy: 0.9514
KNN | Size: 410 | Accuracy: 0.9571
KNN | Size: 460 | Accuracy: 0.9586
KNN | Size: 510 | Accuracy: 0.9571
KNN | Size: 560 | Accuracy: 0.9586
KNN | Size: 610 | Accuracy: 0.9571
KNN | Size: 660 | Accuracy: 0.9600
KNN | Size: 710 | Accuracy: 0.9600
KNN | Size: 760 | Accuracy: 0.9614
```

```
KNN | Size: 810 | Accuracy: 0.9586
KNN | Size: 860 | Accuracy: 0.9586
KNN | Size: 910 | Accuracy: 0.9571
KNN | Size: 960 | Accuracy: 0.9571
KNN | Size: 1010 | Accuracy: 0.9571
KNN | Size: 1060 | Accuracy: 0.9586
KNN | Size: 1110 | Accuracy: 0.9614
KNN | Size: 1160 | Accuracy: 0.9629
KNN | Size: 1210 | Accuracy: 0.9643
KNN | Size: 1260 | Accuracy: 0.9629
KNN | Size: 1310 | Accuracy: 0.9629
KNN | Size: 1360 | Accuracy: 0.9586
KNN | Size: 1410 | Accuracy: 0.9600
KNN | Size: 1460 | Accuracy: 0.9614
KNN | Size: 1510 | Accuracy: 0.9614
KNN | Size: 1560 | Accuracy: 0.9614
KNN | Size: 1610 | Accuracy: 0.9643
KNN | Size: 1660 | Accuracy: 0.9629
KNN | Size: 1710 | Accuracy: 0.9614
KNN | Size: 1760 | Accuracy: 0.9586
KNN | Size: 1810 | Accuracy: 0.9586
KNN | Size: 1860 | Accuracy: 0.9600
KNN | Size: 1910 | Accuracy: 0.9586
KNN | Size: 1960 | Accuracy: 0.9586
KNN | Size: 2010 | Accuracy: 0.9586
KNN | Size: 2060 | Accuracy: 0.9586
KNN | Size: 2110 | Accuracy: 0.9586
KNN | Size: 2160 | Accuracy: 0.9586
KNN | Size: 2210 | Accuracy: 0.9586
KNN | Size: 2260 | Accuracy: 0.9586
KNN | Size: 2310 | Accuracy: 0.9586
KNN | Size: 2360 | Accuracy: 0.9586
KNN | Size: 2410 | Accuracy: 0.9586
KNN | Size: 2460 | Accuracy: 0.9586
KNN | Size: 2510 | Accuracy: 0.9586
KNN | Size: 2560 | Accuracy: 0.9586
KNN | Size: 2610 | Accuracy: 0.9586
KNN | Size: 2660 | Accuracy: 0.9586
KNN | Size: 2710 | Accuracy: 0.9586
KNN | Size: 2760 | Accuracy: 0.9586
```

Final evaluation

In [15]:
```python
# Get the model with best accuracy by choosing the maximum index 1 in the nested
best_model = max(models, key = lambda x: x[1])
print(f"{best_model[0]} | Size: {best_model[2]} | Accuracy: {best_model[1]}")
```

```
SVC(C=11) | Size: 1360 | Accuracy: 0.9728571428571429
```

This is our best Model:

SVC with C = 11

Size on dataset 1360

With an Accuracy 97.29%,

The accurcy will get higher on the kaggle submission.

Kaggle submission

In [16]:
```
# Our best model.
model = best_model[0]

# Getting our test data and dropping useless features.
df_test = pd.read_csv("./assets/test.csv", index_col = 0)
df_test = df_test.drop(columns = useless_features)

# Scaling
X_test2_scaled = scaler.transform(df_test)

# Taken from the kaggle submission side.
y_test2 = model.predict(X_test2_scaled)
y_test2 = pd.DataFrame(y_test2, columns=["Quality"])
y_test2.index.name = "ID"
y_test2[['Quality']].to_csv("submission.csv")
```