AIML

CLASSIFICATION

(predicting water quality)

By: Yongjie(2342377) DAAA/FT/1B/01

Background Information

- This is a dataset collected from an environmental company.
- The dataset contains various information on water from different sources.
- From the quality control perspective, the company wants to build a machine learning model to predict water quality based on the water properties.
- The data set contains 10 columns and 3276 data points
- Potential reasons for poor water quality could be improper sewage and solids disposal, excessive use of pesticides and fertilizers

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalomethanes	Clarity	Quality
0	NaN	204.890456	20791.31898	7.300212	368.516441	564.308654	10.379783	86.990970	2.963135	0
1	3.716080	129.422921	18630.05786	6.635246	NaN	592.885359	15.180013	56.329076	4.500656	0
2	8.099124	224.236259	19909.54173	9.275884	NaN	418.606213	16.868637	66.420093	3.055934	0
3	8.316766	214.373394	22018.41744	8.059332	356.886136	363.266516	18.436525	100.341674	4.628771	0
4	9.092223	181.101509	17978.98634	6.546600	310.135738	398.410813	11.558279	31.997993	4.075075	0

Objective: Create a model to predict water quality

Exploratory Data Analysis

Information on our features

- ph - measures alkalinity/ acidity of water on a scale of 0-14 from very alkaline to very acidic, with 7 being neutral
- Hardness - concentration of minerals in water
- Solids - total amount of solids dissolved and suspended in water
- Chloramines - the total amount of chlorine compound in the water
- Sulfate - total amount of sulfate ions in water
- Conductivity- conductivity of the water
- Organic carbon - the proportion of organic carbon in water
- Trihalomethanes - the measure of chemical compounds formed from the reaction of organic/inorganic materials with chlorine
- Clarity - Measure of transparency of water
- Quality - an overall measure of the quality of water based on properties either 1 or 0

EDA - Getting Descriptive Data

Creating a copy of the data set

```
df_explore = df.copy()
```

```
df_explore.shape
```

```
(3276, 10)
```

Looking at the shape of our data set.

```
df_explore.isna().sum()
```

```
Sulfate              781
ph                   491
Trihalomethanes      162
Hardness               0
Solids                 0
Chloramines            0
Conductivity           0
Organic_carbon         0
Clarity                0
Quality                0
dtype: int64
```

Looking at the number of rows with missing data from the column

Looking at the datatypes of our data

```
df_explore.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3276 entries, 0 to 3275
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   ph               2785 non-null   float64
 1   Hardness         3276 non-null   float64
 2   Solids           3276 non-null   float64
 3   Chloramines      3276 non-null   float64
 4   Sulfate          2495 non-null   float64
 5   Conductivity     3276 non-null   float64
 6   Organic_carbon   3276 non-null   float64
 7   Trihalomethanes  3114 non-null   float64
 8   Clarity          3276 non-null   float64
 9   Quality          3276 non-null   int64
dtypes: float64(9), int64(1)
memory usage: 256.1 KB
```

Summary

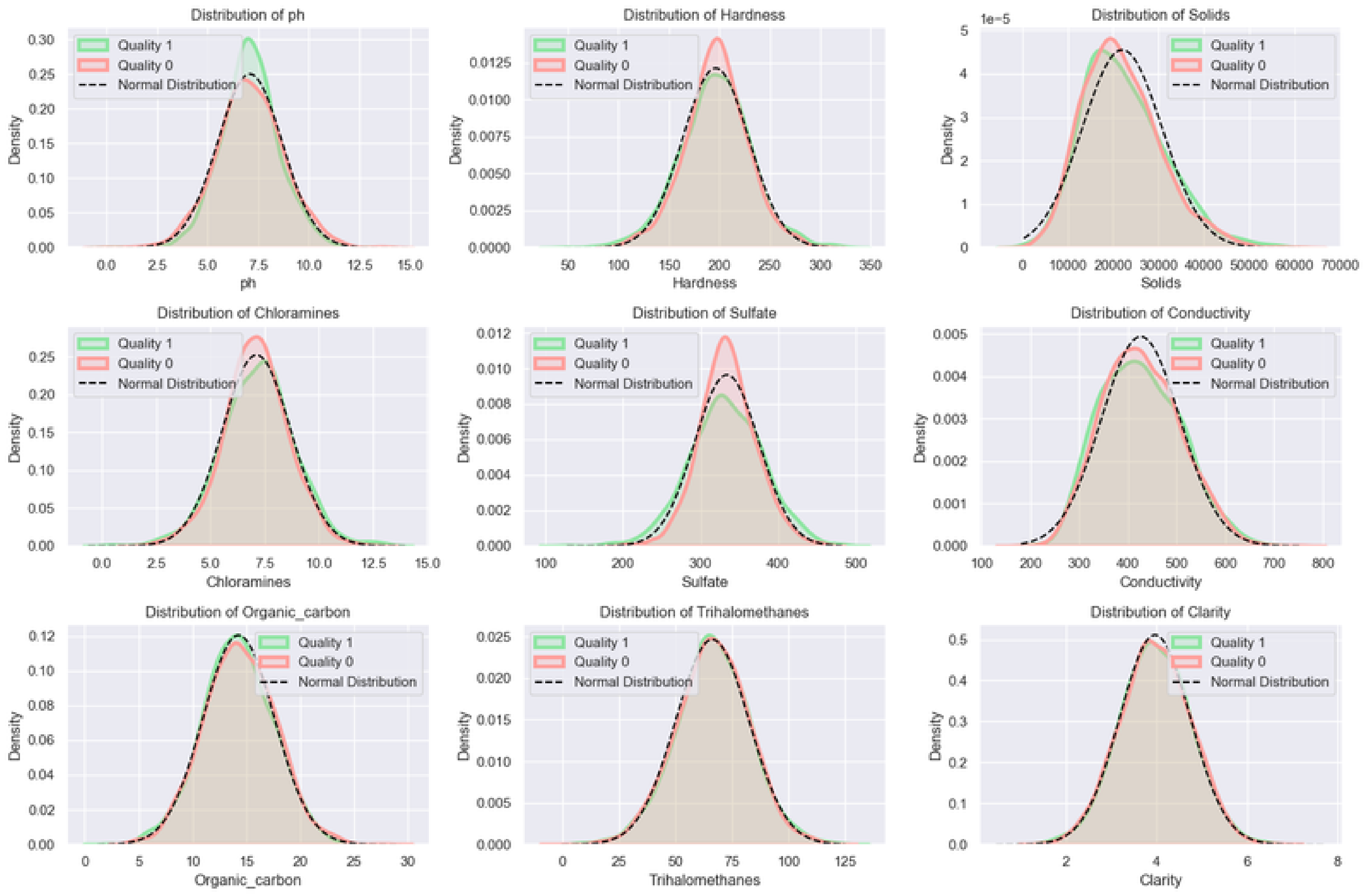
- Working with with contains 3276 rows and 10 columns (9 features + 1 target).
- all the datatype are numeric and that we have no categorical data
- There are missing values for 3 of the columns
- Missing data includes, ph - 17.6%, Sulfate - 31.3% and Trihalomethanes - 5.2%

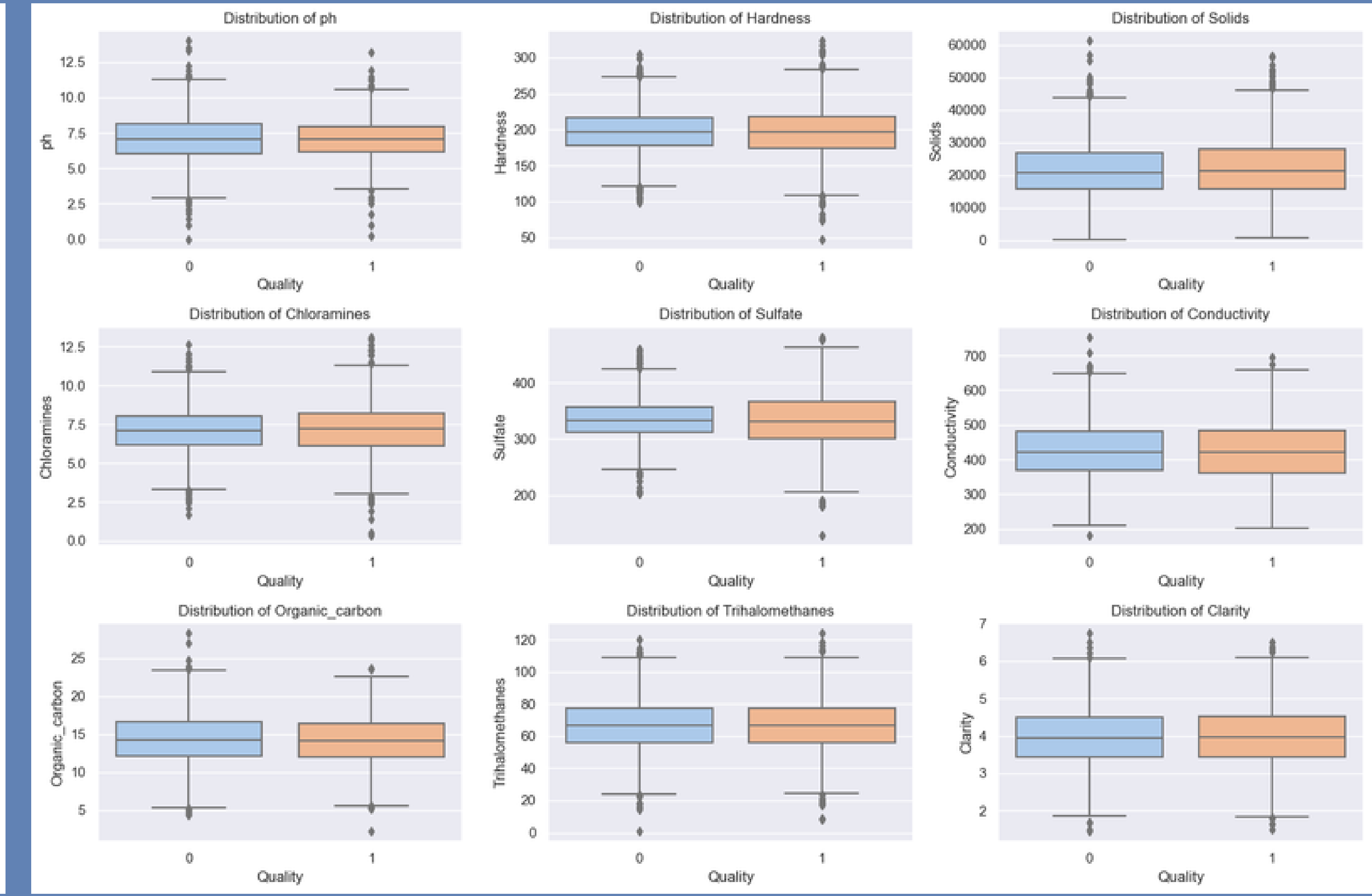
EDA - Target

1.5 times more Quality 0 water compared to Quality 1 water, which although is quite similar, it could be distributed more equally.

Since we are not able to assume what the value of 0 and 1 represents, therefore the goal of our model will be able to best predict the value of 0 and 1 as accurately as possbile rather than predicting one value better.

EDA - Univariate Analysis

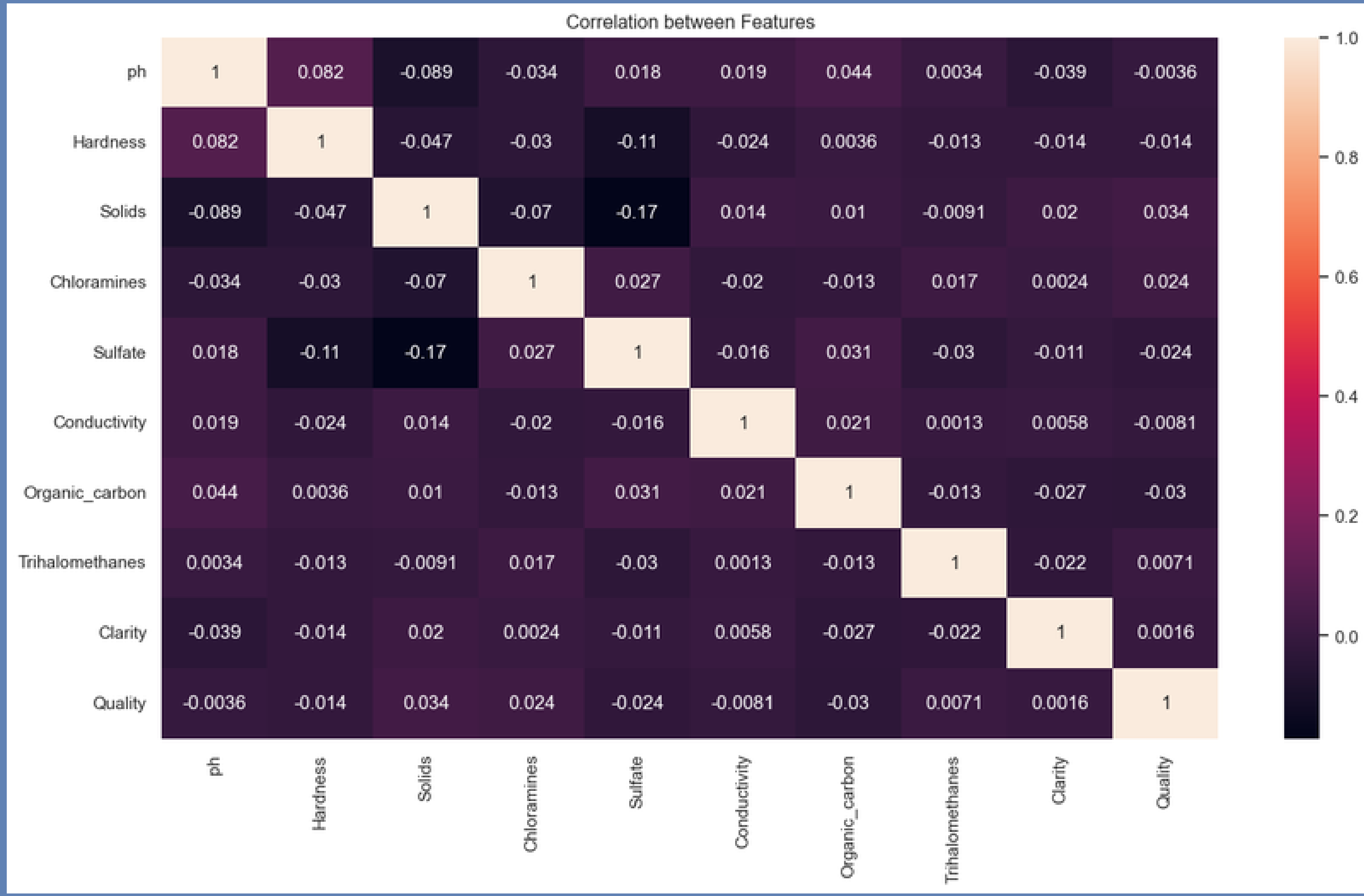


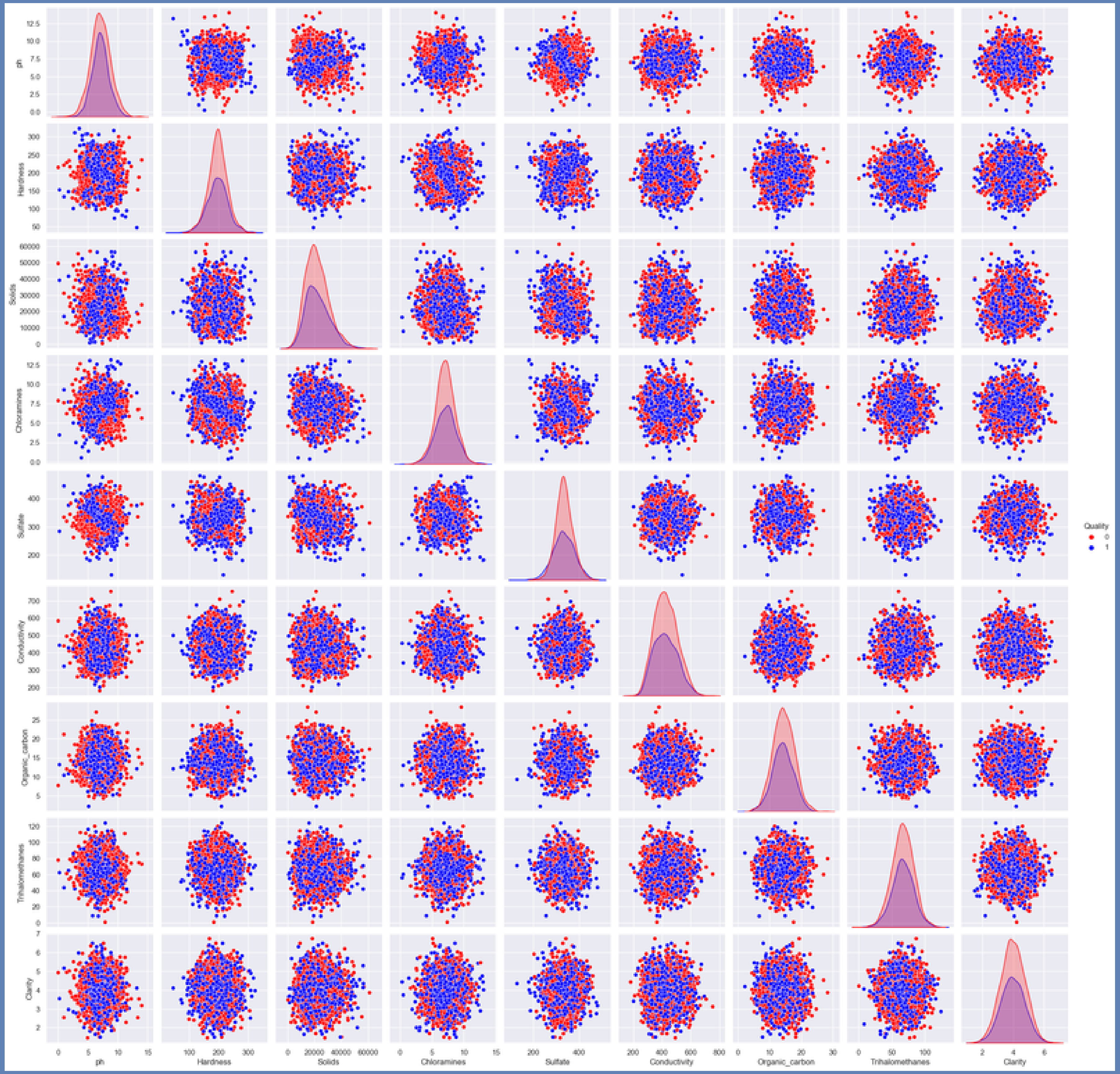


- For the featurese "ph," "Hardness," "Chloramines," "Conductivity," "Organic_carbon," "Trihalomethanes," and "Clarity" they are similar in showing a normal distribution for both quality 0 and 1. We can observe that the graphs for both quality are almost matching, suggesting that for most value of the feature it could be either quality 0 or 1.
- "Sulfate" shows a small difference of 0.004 in density at the median between Quality 0 and Quality 1.
- "Solids" seems to have a right-skewed distribution, suggesting potential outliers.

- Using IQR, I calculated the number of outliers for each column, and got the number of rows with outliers and found that almost 10% of rows have a least 1 outlier in a column
- We can see from the boxplot that there are outliers for every feature in our dataset with almost 10% of rows having outliers, it is important to consider how to deal with it, as removing 10% of dateset could be a crucial mistake as those outliers could represent a new trend and we would also have less datepoints to train our model on.

EDA - Bivariate Analysis





- From the heatmap, we can conclude that there is very less correlation between any of the variables as correlation value for all are close to 0 with the highest value being 0.082 between ph and hardness while the lowest value is -0.17 for sulfate and solids.
- From the pairplot, we can see that there are no obvious trends between the features which is expected as there are weak correlation as seen from the heat map

Data Preparation

1 group data into target and feature

```
X, y = df.drop(['Quality'], axis=1), df['Quality']
```

2 splitting data into train test set

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,stratify=y, random_state=42)
X_training = X_train.copy()
```

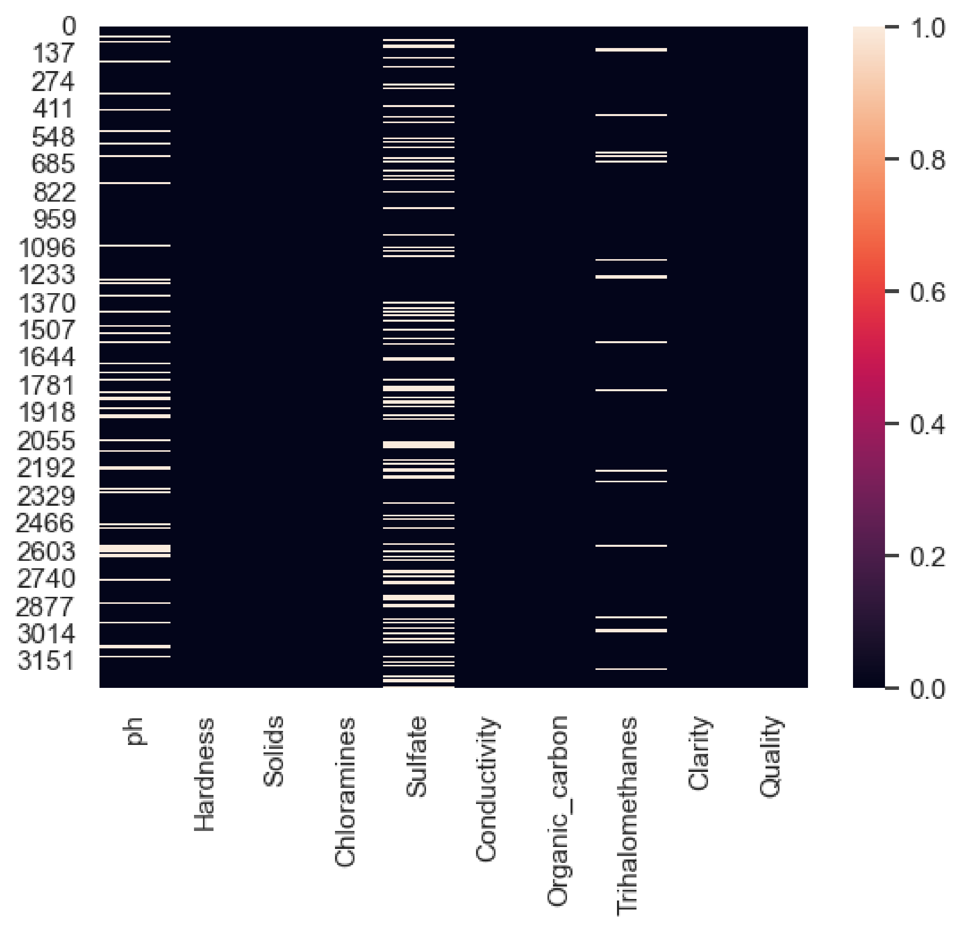
3 impute missing data

There are several ways to impute missing data

- Dropping rows or columns with missing values
- Forward/Backward fill
- Simple imputer using average(mean, mode, or median)
- KNN/Iterative Impute

```
# select columns that need to be imputed
cols = ['ph', 'Sulfate', 'Trihalomethanes']

# init the imputer
imputer = KNNImputer()

# impute the copy of training set to check if imputer is working
X_training.loc[:,cols] = imputer.fit_transform(X_training[cols])
X_test.loc[:,cols] = imputer.fit_transform(X_test[cols])
```

```
ph                 0
Hardness           0
Solids             0
Chloramines        0
Sulfate            0
Conductivity       0
Organic_carbon     0
Trihalomethanes    0
Clarity            0
dtype: int64
```



Data Preparation

4 feature engineering

After doing some research online, I found that there is a linear relationship between solids and conductivity based on the formula:

$$\mathrm{TDS}\ (\mathrm{mg/L}) = k_e \times \mathrm{EC}\ (\mu\mathrm{S/cm})$$

TDS = Solids, CE = Conductivity, and k = constant of proportionality

I also found a safety guideline for ph, according to WHO the drinkability of water is between the range of 6.5 and 8.5. Therefore, we will be creating 2 new features from existing features, k_constant and ph_drinkable, in order to help train our model better.

```python
def k_constant(df):
    df["k_constant"] = pd.to_numeric(df['Solids']/df['Conductivity'])
    return df

def ph_drinkable(df):
    df['ph_drinkable'] = df['ph'].apply(lambda x: 1 if x >= 6.5 and x <=8.5 else 0)
    return df

def createCols(df):
    return ph_drinkable(k_constant(df))

X_training = createCols(X_training)
X_training
```

k_constant	ph_drinkable
44.370112	1
42.393514	0
80.083330	1
58.629762	1
35.467149	1
...	...
23.310432	1
42.671202	0
82.357854	1
45.644156	1
48.793465	1

5 balancing target distribution

There are several ways to resample the data:

- Random Under-Sampling
- Random Over-Sampling
- Random Combined-Sampling
- SMOTE

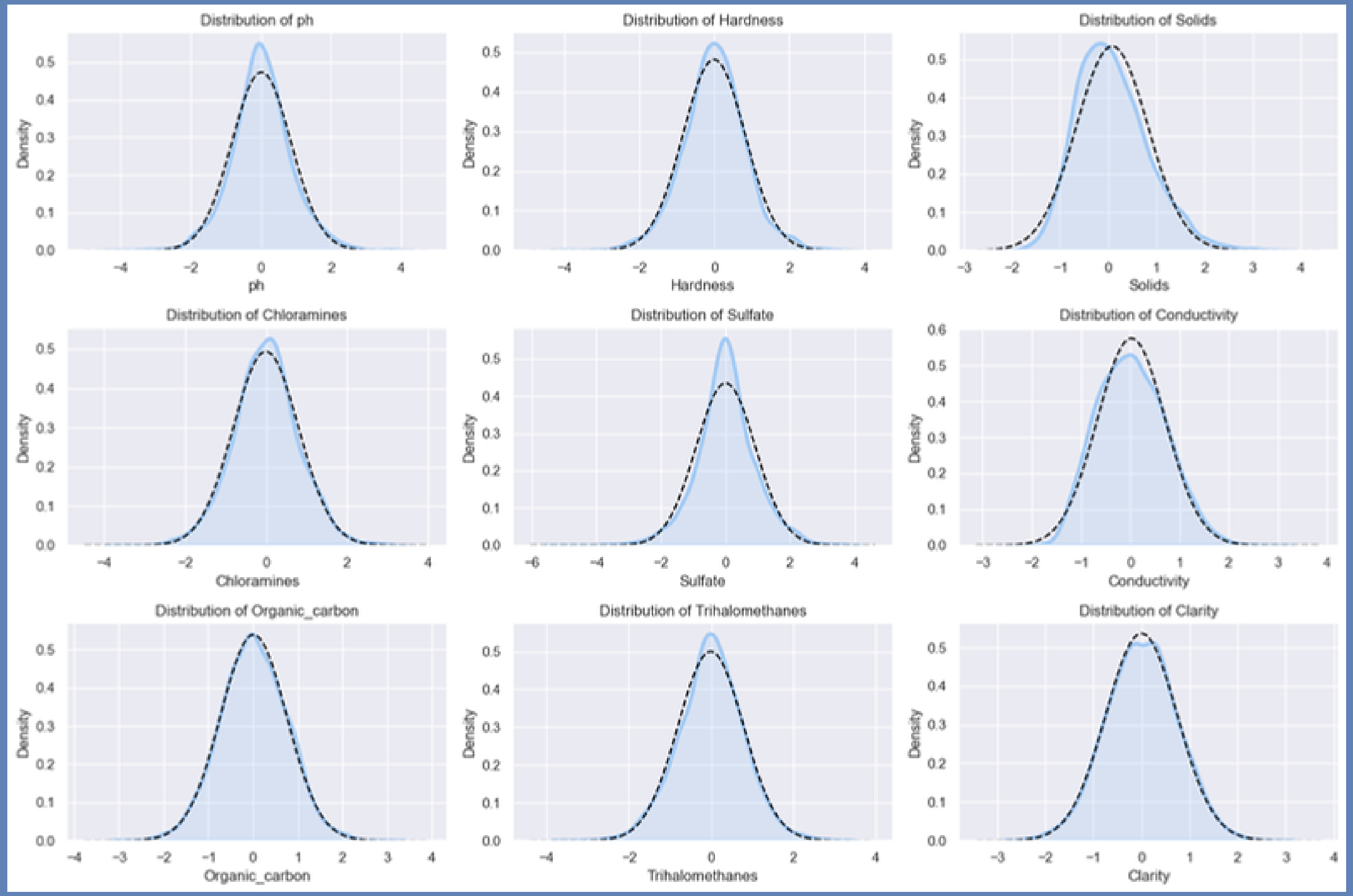
Since we have are dealing with a data set with about 3000, we will be using SMOTE(Synthetic Minority Oversampling Technique) as it increase the number of instance in the minority class(Quality 1) using synthetic data from the dataset.

```python
resample = SMOTE(random_state=69)
X_resampled, y_resampled = resample.fit_resample(X_training, y_train)
```

Before:

After:

Data Preparation



6 standardization

There are several ways we can Standardize our data such as using:

- Standard Scaler
- Robust Scaler

Standard scaler scales the data such that the mean = 0 and standard deviation = 1.
It divides each value by the standard deviation of the feature and subtracts the mean of the feature from each feature:

$$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$

Meanwhile, Robust Scaler removes the median and scales the data according to the Interquartile Range (IQR) which is between 25% and 75%. It uses IQR and median rather than mean and standard deviation:

$$X_{\text{scaled}} = \frac{X - Q_1}{Q_3 - Q_1}$$

Since we have found out during EDA that we need to consider that we are dealing with multiple outliers, we have to take into consideration that the outliers would have a large effect on the standard deviation of the data. Hence, we will use Robust Scaler which is more suitable as it is less influenced by outliers since it uses the IQR.

```
col = list(X_training.columns)

scale = RobustScaler()

X_scaled = X_resampled.copy()
X_scaled[col] = scale.fit_transform(X_scaled[col])
X_scaled
```

Model-Selection

Creating the Pipeline

We will now create a pipeline that helps us to prevent data leakage as imputation and encoding are applied separately to test and train the dataset. It also helps prevent data contamination while allowing us to see the steps taken more easily and reproduce the steps for different models.

```
def createCols(df):
    df = pd.DataFrame(df, columns=['ph', 'Hardness', 'Solids', 'Chloramines', 'Sulfate
    df["k_constant"] = pd.to_numeric(df['Solids']/df['Conductivity'])

    df['ph_drinkable'] = df['ph'].apply(lambda x: 1 if x >= 6.5 and x <=8.5 else 0)
    return df

features_transformer = FunctionTransformer(createCols)

steps = [
    ("impute", KNNImputer()),
    ('preprocessor', features_transformer),
    ("rescale", SMOTE(random_state=69)),
    ("standardize", RobustScaler()),
    ("model")
]
```

Define the scoring metrics

```
scoring_methods = ["accuracy", "precision", "recall", "f1", "roc_auc"]
```

Before creating our models, we will set scoring methods. To score our models and compare the results, we will be using the following common scoring methods:

- Accuracy:
  Proportion of correctly classified datapoints out of the total datapoints (Good when classes are balanced)
- Precision:
  Proportion of true positives out of the total predicted true positives (Important when the cost of false positive is high)
- Recall:
  Proportion of true positives out of the total actual positives (Important when the cost of false negative is high)
- F1 Score:
  Mean of precision and recall (Balance of both precision and recall)
- ROC_AUC (Area Under Curve):
  Measure area under the curve, plots true positive rate against false positive rate (Good for imbalanced datasets)

We will be using StratifiedKFold to cross-validate our model. It creates subsets from the train data depending on the number of folds and cross-validates the subsets with one another.

Model-Selection

Creating baseline dummy model

The dummy model acts as a control, it is a baseline model for comparison to our other models. I will be using DummyClassifier to establish a baseline accuracy score against which is used to compare the performance of more complex models. We will set our parameter strategy to uniform, this will enable to classifier to predict classes uniformly at random regardless of the distribution of the the target. Therefore it will predict quality 1 and quality 0 with a probability of 0.5 for both classes.

```python
steps[STEPS_LEN] = ("model", DummyClassifier(strategy="uniform"))
dummy = IMBPipeline(steps=steps)
dummy.fit(X_train, y_train)

kf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
print(f"Baseline Accuracy Score :{dummy.score(X_test,y_test)}")
scores = cross_validate(
    dummy,
    X_train,
    y_train,
    cv=kf,
    scoring=scoring_methods,
    n_jobs=-1,
    return_train_score=True,
)

# Concatenate along rows (axis=0) to add the mean row
dummy_model_score = pd.concat([pd.DataFrame(scores),
                               pd.DataFrame([pd.DataFrame(scores).mean()],
                                            columns=pd.DataFrame(scores).columns, index=["Mean"])]
                              )

display(dummy_model_score.style.apply(
    lambda x: ["background-color: red; color: white" if v else "" for v in x == x.min()]).apply(
        lambda x: ["background-color: green; color: white" if v else "" for v in x == x.max()])
    )
```

	fit_time	score_time	test_accuracy	train_accuracy	test_precision	train_precision	test_recall	train_recall	test_f1	train_f1
0	0.434400	0.092979	0.480916	0.491942	0.369231	0.383389	0.470588	0.496739	0.413793	0.432765
1	0.448917	0.075475	0.480916	0.491094	0.362903	0.378049	0.441176	0.471739	0.398230	0.419729
2	0.438696	0.103928	0.503817	0.500424	0.401408	0.393388	0.558824	0.517391	0.467213	0.446948
3	0.444837	0.104804	0.480916	0.522053	0.369231	0.409765	0.470588	0.510870	0.413793	0.454765
4	0.463927	0.082510	0.515267	0.510602	0.407407	0.399657	0.539216	0.506522	0.464135	0.446788
5	0.455838	0.146024	0.526718	0.496607	0.401786	0.390304	0.441176	0.516304	0.420561	0.444548
6	0.435450	0.086947	0.477099	0.494911	0.347826	0.385847	0.392157	0.497826	0.368664	0.434741
7	0.455927	0.093995	0.488550	0.516964	0.370968	0.406650	0.450980	0.518478	0.407080	0.455805
8	0.443982	0.102024	0.515267	0.493639	0.413043	0.382378	0.553398	0.486398	0.473029	0.428161
9	0.469838	0.162521	0.553435	0.494911	0.441667	0.385135	0.514563	0.496192	0.475336	0.433666
Mean	0.449181	0.105121	0.502290	0.501315	0.388547	0.391456	0.483267	0.501846	0.430183	0.439792

Model-Selection

Choosing our model

Observation

As observed from the table, there isn't a very outstanding model that performs exceptionally well in every scoring metric. Therefore, we will focus on the F1 score and the learning curve to choose our model.

Baseline Dummy scoring for F1, accuracy, and ROC:

F1: 0.43, Accuracy: 0.5, FOC: 0.5

Based on the results, our top 3 performing models are:

- SVC
- Random Forest Classifier
- KNeighborsClassifier

Define our models

```
# Chosing our model to train and compare
models = [
    ("KNeighborsClassifier", KNeighborsClassifier()),
    ("SVC", SVC()),
    ("AdaBoostClassifier", AdaBoostClassifier()),
    ("LogisticRegression", LogisticRegression()),
    ("Perceptron", Perceptron()),
    ("DecisionTreeClassifier", DecisionTreeClassifier()),
    ("RandomForestClassifier", RandomForestClassifier()),
    ("RidgeClassifier", RidgeClassifier()),
    ("GradientBoostingClassifier", GradientBoostingClassifier()),
    ("RidgeClassifierCV", RidgeClassifierCV()),
    ("ExtraTreesClassifier", ExtraTreesClassifier()),
    ("GaussianNB", GaussianNB()),
]
```

Selecting our model

Looking at our top 3 models with F1 scores of about 0.56, 0.53, and 0.52 respectively, comparing this to our Dummy Model F1 score (around 0.43), we can see that our models performed slightly better by 0.09-0.13.

Examining the learning curve, we can see that the training and cross-validation scores converge to a point of stability. This is good as it indicates that the model performance on unseen data is consistent with its performance on the training data, suggesting that the model is well-balanced without overfitting the provided data. We could further improve this by hyper-tuning our models, but a closer look at these models is necessary.

Scoring table

	fit_time	score_time	test_accuracy	train_accuracy	test_precision	train_precision	test_recall	train_recall	test_f1
SVC	0.871300	0.221107	0.645802	0.742748	0.542509	0.658374	0.590986	0.710268	0.564579
RandomForestClassifier	3.210444	0.144610	0.649618	1.000000	0.554590	1.000000	0.505892	1.000000	0.528112
KNeighborsClassifier	0.455688	0.149471	0.602672	0.763656	0.493168	0.667437	0.567476	0.785605	0.526688
GradientBoostingClassifier	3.590551	0.106216	0.612977	0.772986	0.503975	0.709711	0.540177	0.708309	0.520506
ExtraTreesClassifier	1.198135	0.174752	0.646565	1.000000	0.556085	1.000000	0.462869	1.000000	0.503867
DecisionTreeClassifier	0.561280	0.096297	0.574046	1.000000	0.459688	1.000000	0.528460	1.000000	0.491318
GaussianNB	0.444276	0.096574	0.561832	0.573325	0.442074	0.455342	0.468627	0.477719	0.453976
AdaBoostClassifier	1.184364	0.131850	0.542366	0.618024	0.421432	0.510337	0.470626	0.533599	0.443788
Perceptron	0.452245	0.096284	0.511832	0.511238	0.402761	0.401119	0.493223	0.488248	0.427457
LogisticRegression	0.442799	0.100582	0.479771	0.510390	0.364004	0.394926	0.452161	0.480764	0.402803
RidgeClassifier	0.463354	0.103546	0.479389	0.510136	0.363535	0.394768	0.451180	0.481090	0.402137
RidgeClassifierCV	0.428924	0.097520	0.478244	0.510517	0.362587	0.395085	0.451190	0.481199	0.401538

Learning Curve

Hypertuning

1

In order to hypertune our model we will have to first look at the possible hyperparameters that we can tune.

```
print(list(SVC().get_params().keys()))
print(list(KNeighborsClassifier().get_params().keys()))
print(list(RandomForestClassifier().get_params().keys()))
```

2

Creating parameter grid and using RandomisedSearchCV() to get the best parameter

```
# RFC Hypertune
param_grid = {
    'n_estimators': [50, 100, 200, 300, 400],
    'max_depth': [None, 10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10, 15],
    'min_samples_leaf': [1, 2, 4, 8],
    'bootstrap': [True, False]
}

kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Create model based on pipeline
steps[STEPS_LEN] = (
    "hyper", RandomizedSearchCV(
        RandomForestClassifier(),
        param_grid,
        cv=kf,
        n_jobs=-1,
        verbose=1,
        scoring="f1"
    )
)

rfc_search = IMBPipeline(steps=steps)

# fit model
rfc_search.fit(X_train, y_train)

print(rfc_search["hyper"].best_params_)
print(rfc_search["hyper"].best_score_)
```

3

Taking the best esitmator and fitting it into the model parmater

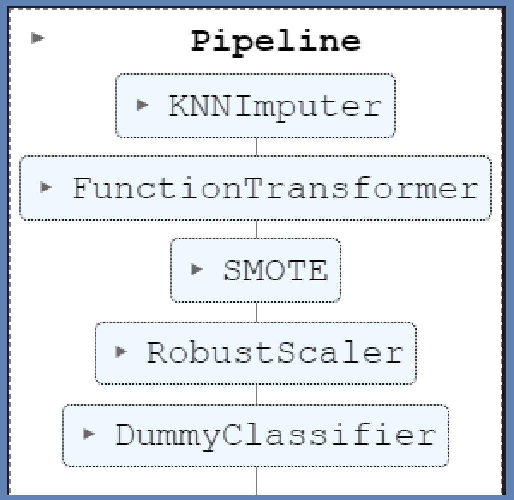
```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
{'n_estimators': 300, 'min_samples_split': 10, 'min_samples_leaf': 2, '
0.7080508500064718
```

```
# creating model based on parameter of best result
svc_model = svc_search["hyper"].best_estimator_
rfc_model = rfc_search["hyper"].best_estimator_
knn_model = knn_search["hyper"].best_estimator_
```

4

Fitting the train data into the model

```
# SVC TUNED
steps[STEPS_LEN] = ("model", svc_model)
svc_clf = IMBPipeline(steps=steps)
svc_clf.fit(X_train, y_train)

# SVC UNTUNED
steps[STEPS_LEN] = ("model", SVC())

svc_clf_untuned = IMBPipeline(steps=steps)
svc_clf_untuned.fit(X_train, y_train)
```



Evaluating hypertuned models

Creating ROC Curve to look at AUC and compare

From the ROC Curve, we can see that only Random Forest Classifier had a slight improvement as its AUC for tuned model was greater than untuned by 0.01, whereas KNN Classifier remained the same while SVC had a drop of 0.09 after hypertuning. Compared to the dummy of 0.5, we can tell that the models are able to distinguish between Quality 0 and 1 better and is less likely to misclarify between the two targets.

Classification Report

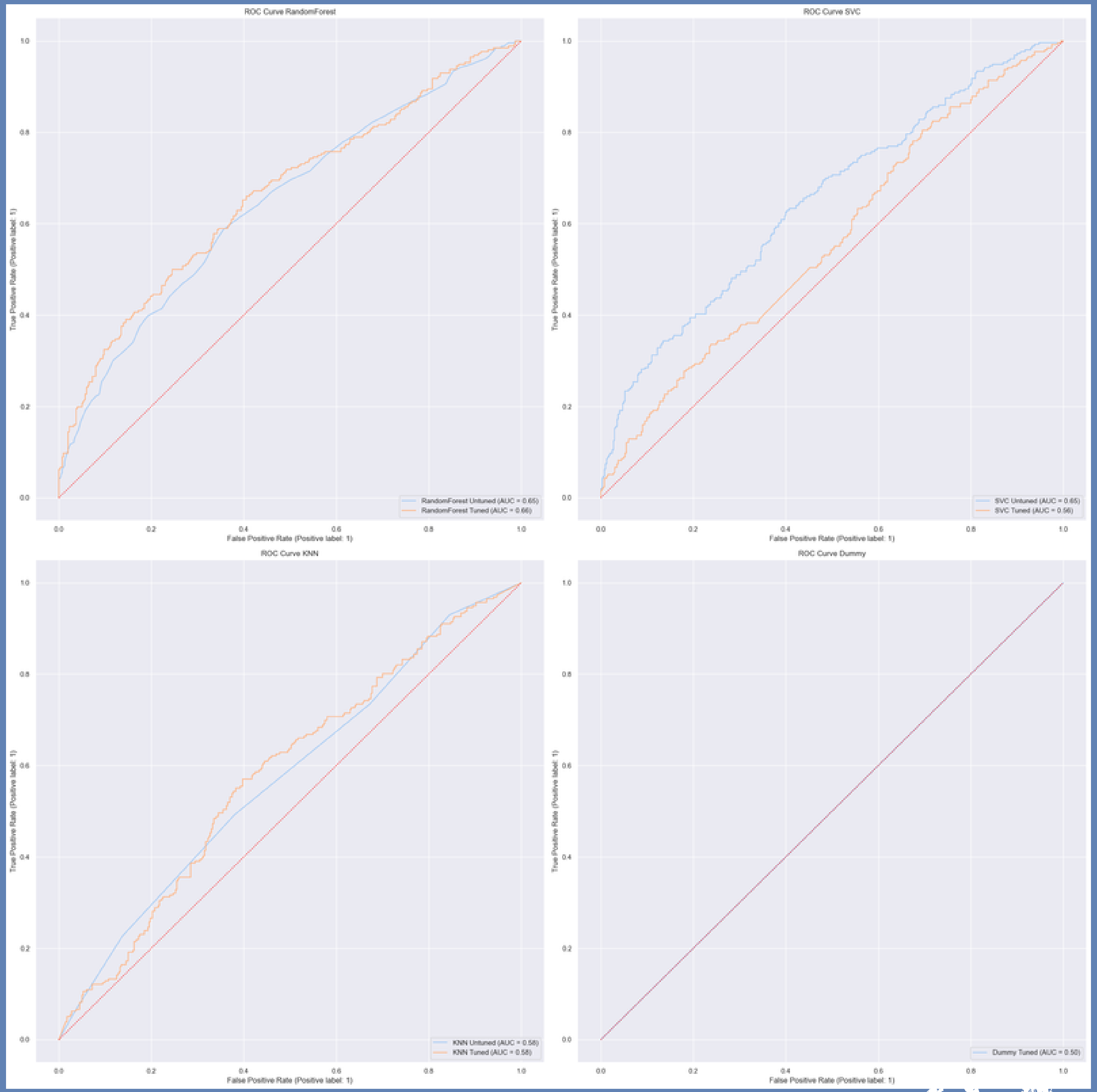
From the classification report, we can see that overall our Random Forest Classifier performed the best with the highest f1 score of 0.73 for 0 and 0.51 for 1 and an overall accuracy of 0.65. Overall, all 3 models out performed the dummy model which had f1 score of 0.56 and 0.48 for 0 and 1 respectively and an accuracy of 0.52.

rfc:	precision	recall	f1-score	suppor
0	0.694	0.770	0.730	400
1	0.566	0.469	0.513	256
accuracy			0.652	656
macro avg	0.630	0.619	0.621	656
weighted avg	0.644	0.652	0.645	656
)				

knn:	precision	recall	f1-score	support
0	0.670	0.635	0.652	400
1	0.473	0.512	0.492	256
accuracy			0.587	656
macro avg	0.572	0.573	0.572	656
weighted avg	0.593	0.587	0.589	656
)				

svc:	precision	recall	f1-score	support
0	0.625	0.632	0.629	400
1	0.414	0.406	0.410	256
accuracy			0.544	656
macro avg	0.520	0.519	0.519	656
weighted avg	0.543	0.544	0.543	656
)				

dummy:	precision	recall	f1-score	support
0	0.639	0.495	0.558	400
1	0.416	0.562	0.478	256
accuracy			0.521	656
macro avg	0.527	0.529	0.518	656
weighted avg	0.552	0.521	0.527	656
)				

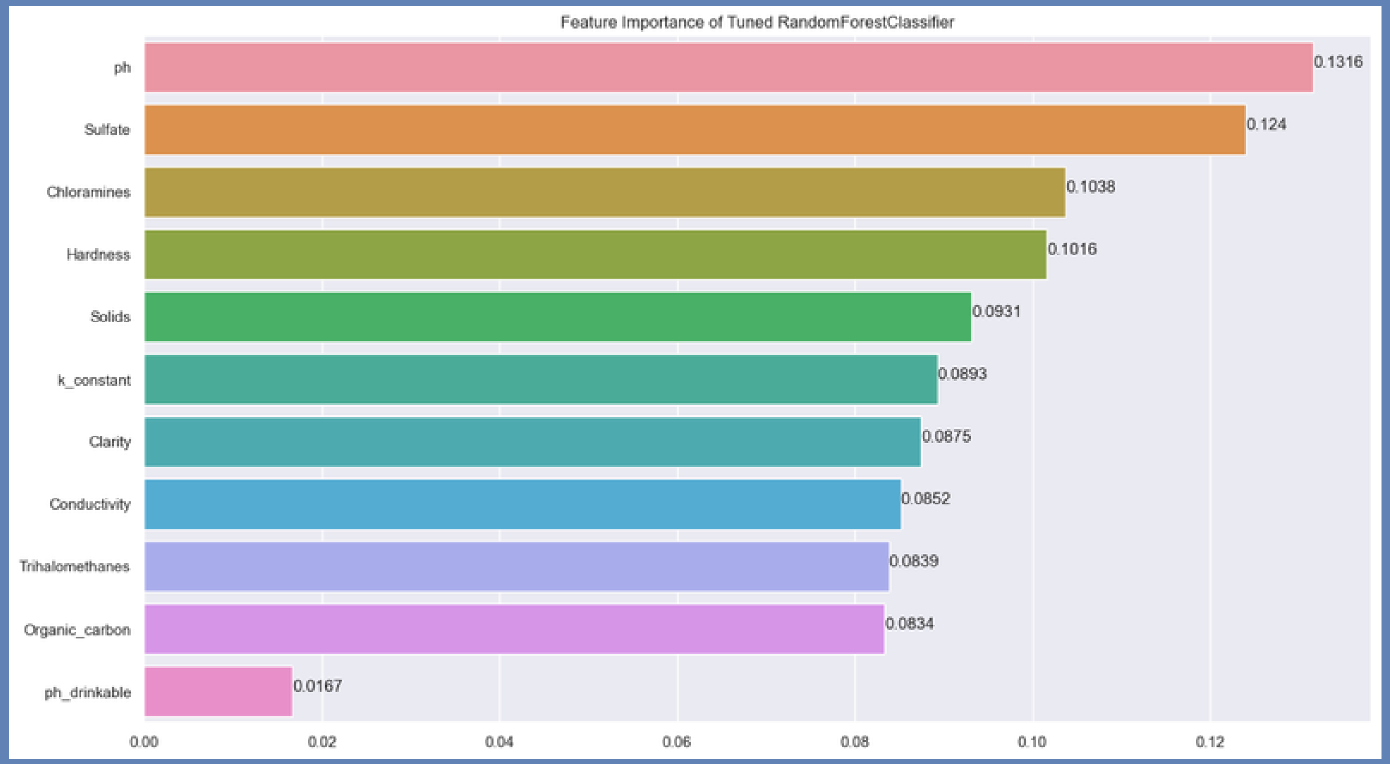


Feature importance

Creating a table of feature importance.

Since our RFC performed the best we will be using it to look at our feature importance, additionally, out of the 3 models only RFC can return feature importance as the other 2 models are distance-based models.

From the table, we can observe that actually, the feature with the most importance is ph while the feature with the least importance is ph_drinkable which is one of our feature-engineered features. Hence, we will now try to drop it instead and see how well our model fair



Comparing dropped col against our previous model

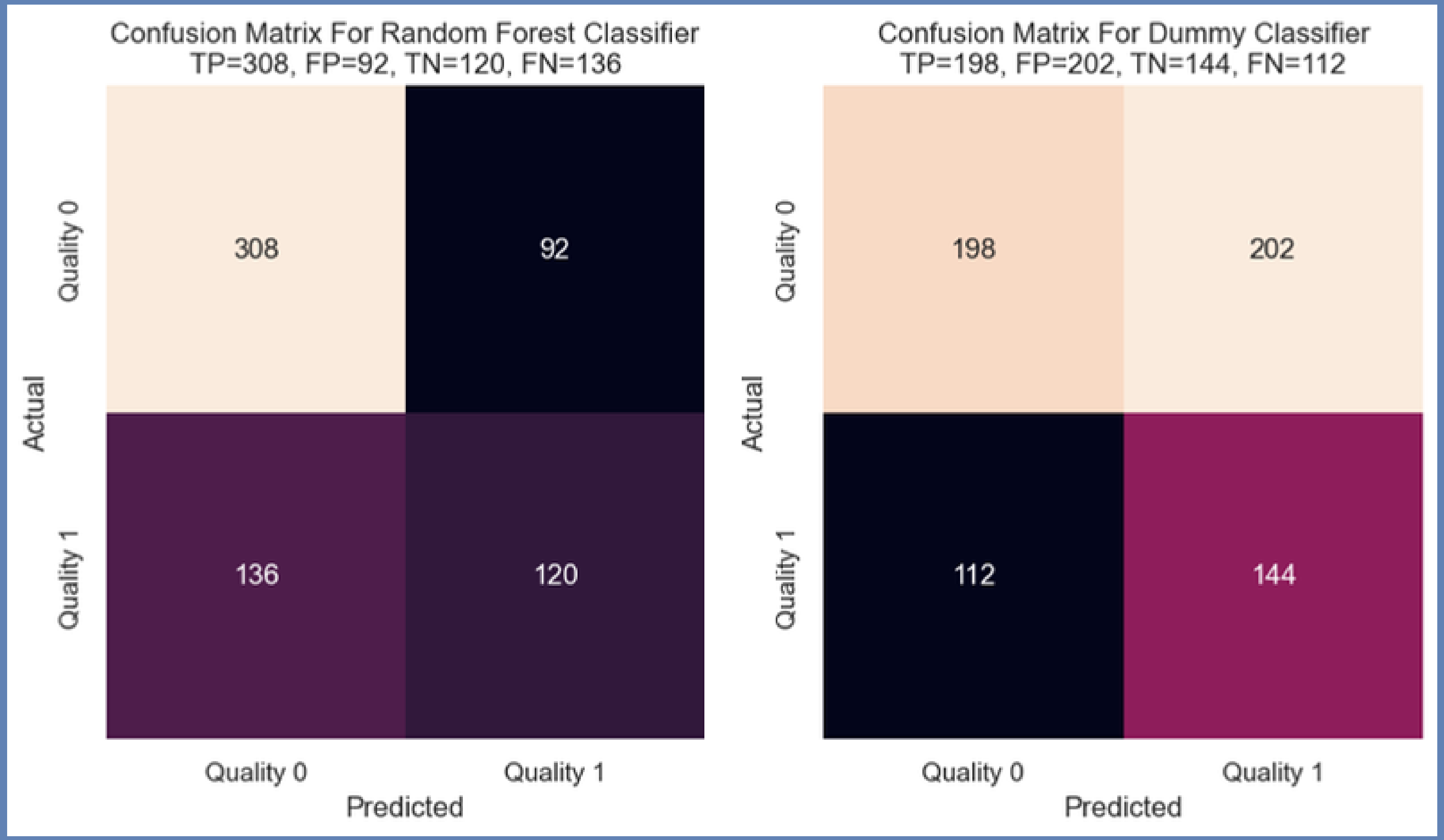
After dropping the ph_drinkable column and training our model and hyper tuning it again, we can observe that the results of our model is still quite similar, but we can see hat the f1-score for both 0 and 1 had decreased slightly by 0.01

	precision	recall	f1-score	support
0	0.69	0.77	0.72	400
1	0.55	0.45	0.50	256
accuracy			0.64	656
macro avg	0.62	0.61	0.61	656
weighted avg	0.63	0.64	0.63	656

```
def createCols(df):
    df = pd.DataFrame(df, columns=['ph', 'Hardness', 'Solids',
    df["k_constant"] = pd.to_numeric(df['Solids']/df['Conductiv
    return df

features_transformer = FunctionTransformer(createCols)

temp_steps = [
    ("impute", KNNImputer()),
    ('preprocessor', features_transformer),
    ("rescale", SMOTE(random_state=69)),
    ("standardize", RobustScaler()),
    ("model")
]
```

Final Evaluation

As we stated earlier we have chosen to use the Random Forest Classifier that has been hypertuned and will now begin our final evaluation.




```
rfc:            precision    recall  f1-score   suppor

           0      0.694     0.770     0.730      400
           1      0.566     0.469     0.513      256

    accuracy                          0.652      656
   macro avg      0.630     0.619     0.621      656
weighted avg      0.644     0.652     0.645      656
)
```

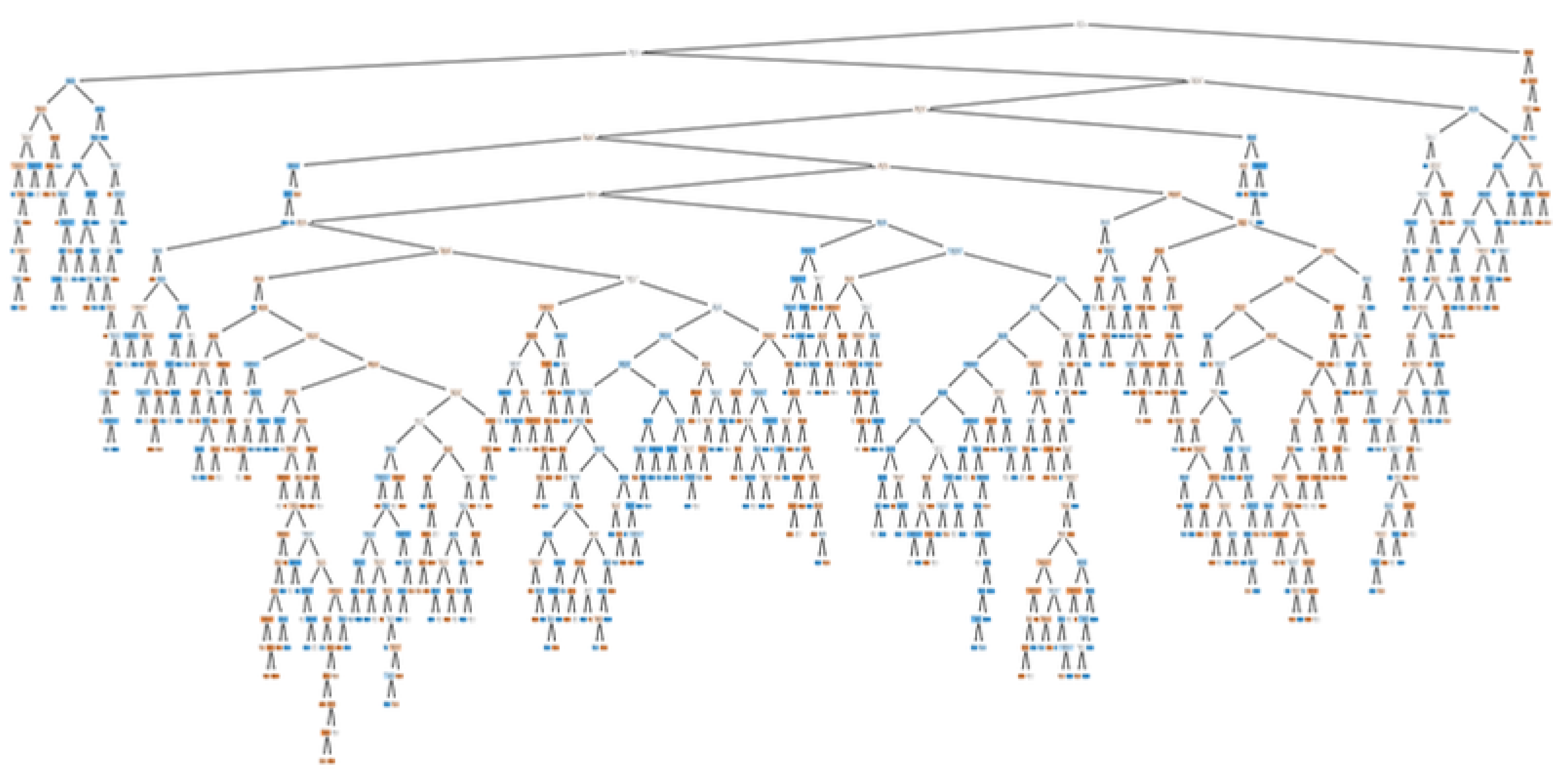
```
dummy:          precision    recall  f1-score   support

           0      0.639     0.495     0.558      400
           1      0.416     0.562     0.478      256

    accuracy                          0.521      656
   macro avg      0.527     0.529     0.518      656
weighted avg      0.552     0.521     0.527      656
)
```

By plotting a transformation matrix we can see from here that our model is better at predicting water quality correctly compared to the dummy model, it predicts 86 more correct cases out of the 656. test cases. Our model predicted water Quality 0 correctly more than 1.5 times more than the dummy while the dummy only predicted Quality 1 correctly by 24 more cases compared to our model.

Looking at the difference in classification report, our model performed better by 0.17, 0.03 for f1 score for 0 and 1 respectively and had a higher accuracy of 0.13

Diagram of our final Model

```
DecisionTreeClassifier(max_depth=40, max_features='sqrt', min_samples_leaf=2,
                       min_samples_split=5, random_state=1051737660)
```



Conclusion

We have successfully created a model to predict water quality andwe can see all features are almost equally important in determining water quality as they all have an impact.

From this project, I learnt how to create a model and doing my own research to better improve the quality of my model.

Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras finibus viverra nisi, tincidunt egestas risus viverra ut. Interdum et malesuada fames ac ante ipsum primis in faucibus. Nulla iaculis iaculis ex non porta. Donec sed tellus erat. Vestibulum congue magna aliquam, iaculis urna ac, semper sem. Nullam at aliquam eros, quis porta velit. Nunc posuere purus lectus.