



Introduction to Data Science

DATA SCIENCE PROJECT

Supervised Models of Data Science with Wine Quality Dataset
By Hsuan (Chih-Hsuan, Wang)



November 2nd, 2021

University of California, Irvine

Division of Continuing Education

INTRODUCTION

There are many things in life that are worth studying in a data science way. For example, we can use population survey data sets to run models to discover social phenomena, or we can build models based on chemical data to find relevant facts that we didn't know before, or even we can reveal the mystery of living creatures according to biological characteristics. There are many things that have certain patterns. If we find these patterns, we can make some predictions based on these patterns. There are so many kinds of knowledge fields in the world, and there are so many different patterns waiting for us to discover. This time we are using the famous wine quality dataset on UCI ML data repository to find the pattern of this dataset.

We will use the wine quality dataset on the UCI ML data repository to complete a data science project. This dataset allows us to run a variety of models. We can use these chemical data to run a clustering model to know how these wines can be distinguished, to run the association model to find whether there is a relationship between specific chemical feature and specific chemical feature, to run the regression model to get the predictions from the data, to run the classification model to understand what kind of pattern can determine the attribution of the category, and even run the neural network model to tell us what this pattern is. Generally speaking, the unsupervised model allows us to find some interesting findings and see if we can use these interesting findings to create commercial value, so it is not meaningful to discuss the unsupervised model here. Therefore, we will make good use of the supervision model here. We will use the quality of wine to run the regression model and the class of wine to run the classification model.

In the field of data science, there are many kinds of libraries with many kinds of models, such as scikit-learn, Keras, XGBoost, etc. Our focus this time will be on scikit-learn, there are many models in scikit-learn, such as Linear Regression, Logistic Regression, Support Vector Machines, Gaussian Processes, Nearest Neighbors, Decision Trees, etc., there are also ensemble methods, in bagging methods there are Random Forests and in boosting methods are AdaBoost, Gradient Tree Boosting, There are even Neural network models. We may not be able to cover all the models, but we will try to run as many models as possible so that we can find models that perform well.

Only running the model cannot be regarded as a complete data science project. We need a certain operation process to help us confirm whether we have a complete data science process. CRISP-DM is a great workflow that can be referenced. CRISP-DM is the most famous framework for defining data science workflows. It describes six stages, business understanding, data understanding, data preprocessing, modeling, evaluation, and deployment. The whole process is reversible. We will follow this workflow step by step to complete our data science project.

In this project, we will use Python to implement data science projects throughout the process. As for my integrated development environment, I will use Jupyter Notebook. Although coding with Visual Studio Code is more convenient and easier to debug, using Jupyter Notebook and convert the coding web pages to PDF files is cleaner and tidy. Jupyter Notebook just like a notebook, it is also more suitable for making notes, drawing key points, etc. In other words, when I attach my code, I think it's more appropriate.

Project implementation method and steps

In the field of data science, there are many standard processes for data science to refer to. Many people think that the best data science process is the "Cross Industry Standard Process for Data Mining" (CRISP-DM). I personally think this is a good standard process in business, because it not only considers the data processing process, but also considers the business purpose of data science. No matter what we are, our most concern is to make money. If you only consider data science, you may not achieve the original purpose of data science. Therefore, we will use CRISP-DM. There are six parts in CRISP-DM: business understanding, data understanding, data preparation, modeling, evaluation, deployment.

Business Understanding

Wine tasting is a kind of enjoyment, just like enjoying the daylight with music. We want to know what kind of wine will make people more enjoyable when tasting. What kind of characteristics or what kind of pattern in the chemical data makes people feel better about the wine?

In this way, we can let manufacturers determine wine quality standards based on these characteristics or dataset patterns and improve the quality of wine according to quality standards, so that manufacturers can produce better wines and benefit more. Where the commercial value is, where the data science value is. And people will have a better wine tasting experience. So, we can also improve the quality of people's enjoyment of life.

Using the dataset created with wine samples, where the features is based on chemical data (like PH values, etc.) and the target variables is based on sensory data. The sensory data is based on experts who score the wine quality between 0 (very poor) and 10 (very good) and calculate the median of 3 evaluations made by wine experts. We can differentiate products based on the quality of these wines and give different prices to maximize profits.

In addition, we can also use these data to find some special phenomena. For example, based on these chemical data, we can know what chemical composition will tell us whether the item is red wine or white wine. For example, the model says it is red wine when it is actually white wine, it may develop new business opportunities. Maybe it can become a new product, white wine that tastes like red wine or red wine that tastes like white wine. The classification task is not only used for classification, but also allows us to find outliers. Sometimes special outliers are more valuable, but if our goal is outliers, please remember not to make outliers more , bur to make the model more accurate. Our main goal is to accurately find outliers, not to find the more outliers.

And use several evaluation methods such as MSE, MAE, accuracy, and confusion matrix to evaluate the quality of the model and improve the model. Finally, based on these evaluation values, find the best model.

Also, we plot the relative importance of the input variables. This plot allows us to understand what the pattern of this model looks like in the dataset. Because we do data science for businesspeople which don't understand these models, so we need to use some plot to explain our model and the result. So that they can use these results to change their business model or manufacturing technology or quality standards or marketing methods, etc.

Data Understanding

UCI Machine Learning Repository has always been a well-known dataset resource, which contains many historical datasets. The wine quality dataset we used comes from UCI Machine Learning Repository. The dataset is a combination of two datasets which related to red and white variants of the Portuguese "Vinho Verde" wine. For more details, the reference [P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis, Viticulture Commission of the Vinho Verde Region(CVRVV), Porto, Portugal 2009]. Due to privacy and logistic issues, only physicochemical (inputs) and sensory (the output) variables are available (e.g. there is no data about grape types, wine brand, wine selling price, etc.).

Dataset URL: <https://archive.ics.uci.edu/ml/datasets/wine+quality>

The first thing we do is import the dataset:

```
dataset=pd.read_csv('Combined_WineQuality_RedWhite.csv')
```

Next, we can take a look at this dataset:

	FixedAcidity	VolatileAcidity	CitricAcid	ResidualSugar	Chlorides	FreeSulfurDioxide	TotalSulfurDioxide	Density	pH	Sulphates	Alcohol	Quality[DV]	Color[DV]
0	6.2	0.270	0.32	6.3	0.048	47.0	159.0	0.99282	3.21	0.60	11.0	6	White
1	7.4	0.635	0.10	2.4	0.080	16.0	33.0	0.99736	3.58	0.69	10.8	7	Red
2	6.3	0.230	0.33	6.9	0.052	23.0	118.0	0.99380	3.23	0.46	10.4	6	White
3	6.2	0.630	0.31	1.7	0.088	15.0	64.0	0.99690	3.46	0.79	9.3	5	Red
4	7.6	0.270	0.52	3.2	0.043	28.0	152.0	0.99129	3.02	0.53	11.4	6	White
...
6492	7.6	0.300	0.37	1.6	0.087	27.0	177.0	0.99438	3.09	0.50	9.8	5	White
6493	6.6	0.120	0.25	1.4	0.039	21.0	131.0	0.99114	3.20	0.45	11.2	7	White
6494	6.6	0.320	0.24	1.3	0.060	42.5	204.0	0.99512	3.59	0.51	9.2	5	White
6495	6.8	0.250	0.27	10.7	0.076	47.0	154.0	0.99670	3.05	0.38	9.0	5	White
6496	6.5	0.270	0.26	11.0	0.030	2.0	82.0	0.99402	3.07	0.36	11.2	5	White

The data set has 13 columns and 6497 rows. Each row in the data set corresponds to a wine. For regression tasks, we use the "quality" column as the target variable, and for classification tasks, we use the "color" column as the target variable. The data is divided into red wine and white wine.

Columns: fixed acidity, volatile acidity, citric acid, residual sugar, chloride, free sulfur dioxide, total sulfur dioxide, density, pH, sulfate, alcohol, quality, color.

Then, we observe the structure of the data:

```
print(dataset.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6497 entries, 0 to 6496
Data columns (total 13 columns):
#   Column                Non-Null Count  Dtype
---  ---
0   FixedAcidity           6497 non-null   float64
1   VolatileAcidity        6497 non-null   float64
2   CitricAcid             6497 non-null   float64
3   ResidualSugar          6497 non-null   float64
4   Chlorides              6497 non-null   float64
5   FreeSulfurDioxide      6497 non-null   float64
6   TotalSulfurDioxide     6497 non-null   float64
7   Density                6497 non-null   float64
8   pH                    6497 non-null   float64
9   Sulphates              6497 non-null   float64
10  Alcohol                6497 non-null   float64
11  Quality[DV]            6497 non-null   int64
12  Color[DV]              6497 non-null   object
dtypes: float64(11), int64(1), object(1)
memory usage: 660.0+ KB
None
```

Data Preprocessing

Step 1: - Check every column:

For a single column, what we have to do is data cleaning: cleaning the messy data into data that we can use (such as data labelling, upper and lower case, or abbreviation, non-abbreviation, etc.). Deal with null values: fill in the missing value of each column or delete entire records. Finally, data standardization: consider the model's assumptions about the data and convert the data into a format in which the model can work better, so that the model can perform better.

From the above results, we know that we do not have any missing values, so we do not need to perform data imputation. But we need convert the non-numeric value into a numeric value and consider whether to standardize the numeric value.

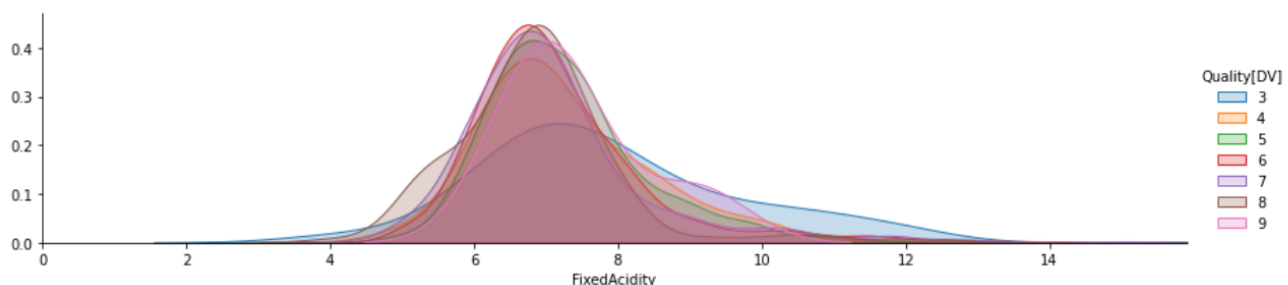
We can use the kdeplot in the library “seaborn” to help us determine whether the column is standardized. Draw a different kdeplot for each column one by one. And make sure that the target variable is correct, the target variable of the regression model is “quality”, and the target variable of the classification model is “color”. By visualizing the data, we can know the status of the data and know how we want to process the data.

Now, we will observe the status of the value of each column one by one:

Column - FixedAcidity:

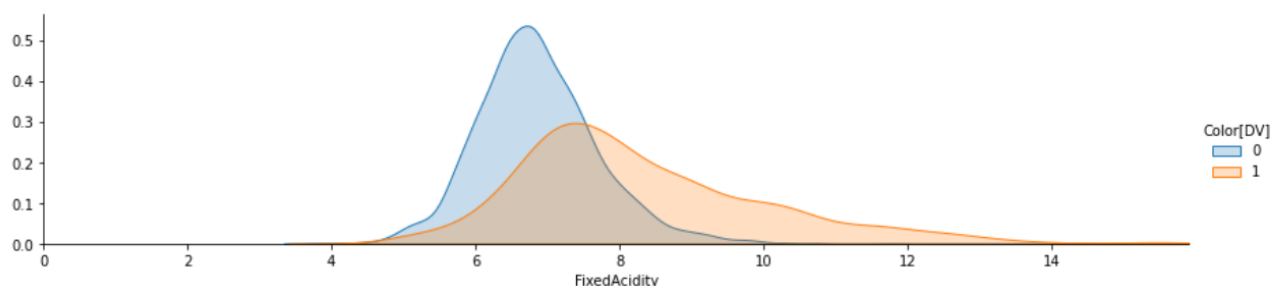
```
#FixedAcidity
facet = sns.FacetGrid(dataset, hue="Quality[DV]", aspect=4)
facet.map(sns.kdeplot, 'FixedAcidity', shade=True)
facet.set(xlim=(0, dataset['FixedAcidity'].max()))
facet.add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x1c44f9fc400>



```
#FixedAcidity
facet = sns.FacetGrid(dataset, hue="Color[DV]", aspect=4)
facet.map(sns.kdeplot, 'FixedAcidity', shade=True)
facet.set(xlim=(0, dataset['FixedAcidity'].max()))
facet.add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x1c453520a30>

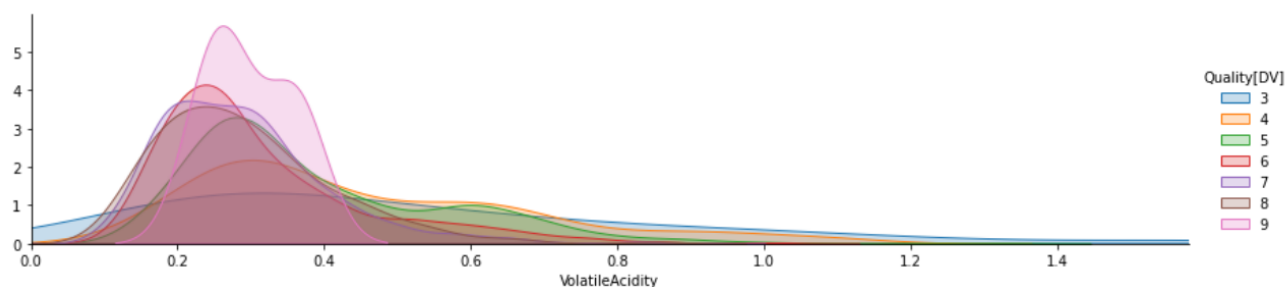


The value in this column seems to be close to the normal distribution for the quality, but the color distribution is a bit to the left, so we can consider the column to be standardized before we run the classification model.

Column - VolatileAcidity:

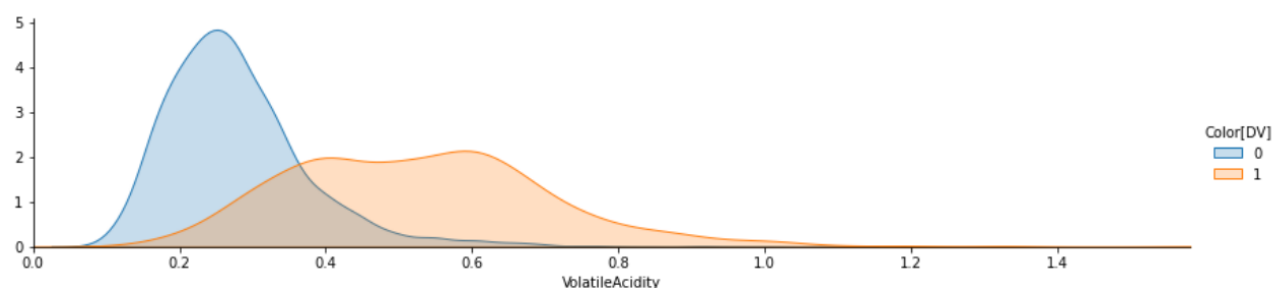
```
#VolatileAcidity
facet = sns.FacetGrid(dataset, hue="Quality[DV]", aspect=4)
facet.map(sns.kdeplot, 'VolatileAcidity', shade= True)
facet.set(xlim=(0, dataset['VolatileAcidity'].max()))
facet.add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x1c4535871c0>



```
#VolatileAcidity
facet = sns.FacetGrid(dataset, hue="Color[DV]", aspect=4)
facet.map(sns.kdeplot, 'VolatileAcidity', shade= True)
facet.set(xlim=(0, dataset['VolatileAcidity'].max()))
facet.add_legend()
```

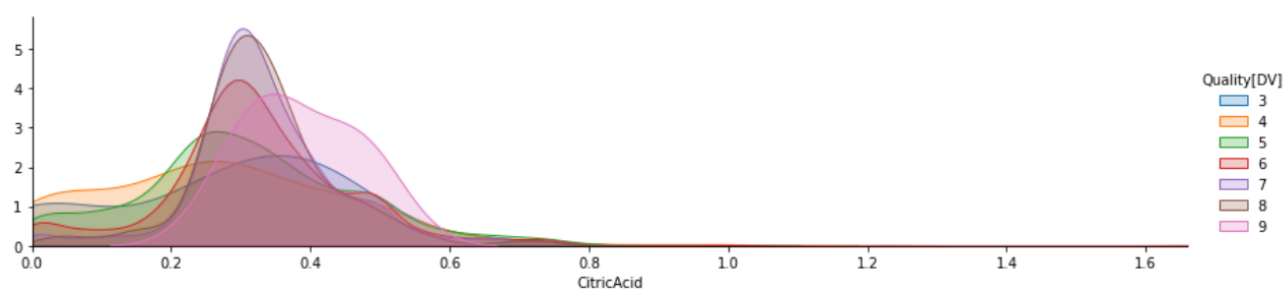
<seaborn.axisgrid.FacetGrid at 0x1c4535ca040>



Column – CitricAcid:

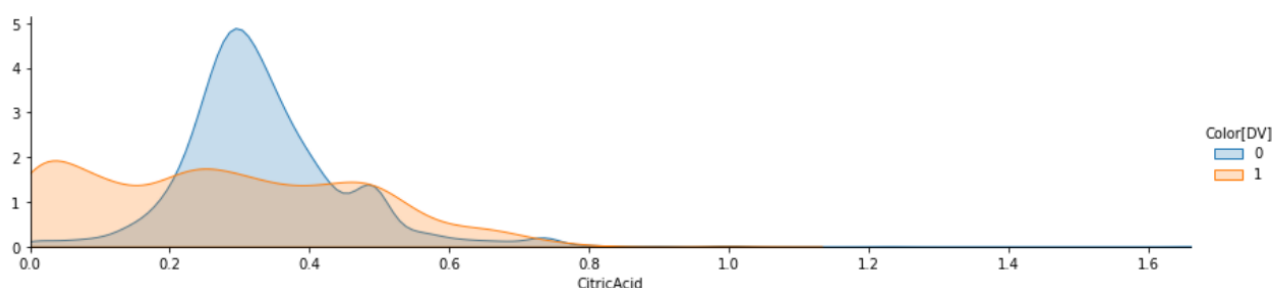
```
#CitricAcid
facet = sns.FacetGrid(dataset, hue="Quality[DV]", aspect=4)
facet.map(sns.kdeplot, 'CitricAcid', shade= True)
facet.set(xlim=(0, dataset['CitricAcid'].max()))
facet.add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x1c4536581c0>



```
#CitricAcid
facet = sns.FacetGrid(dataset, hue="Color[DV]", aspect=4)
facet.map(sns.kdeplot, 'CitricAcid', shade= True)
facet.set(xlim=(0, dataset['CitricAcid'].max()))
facet.add_legend()
```

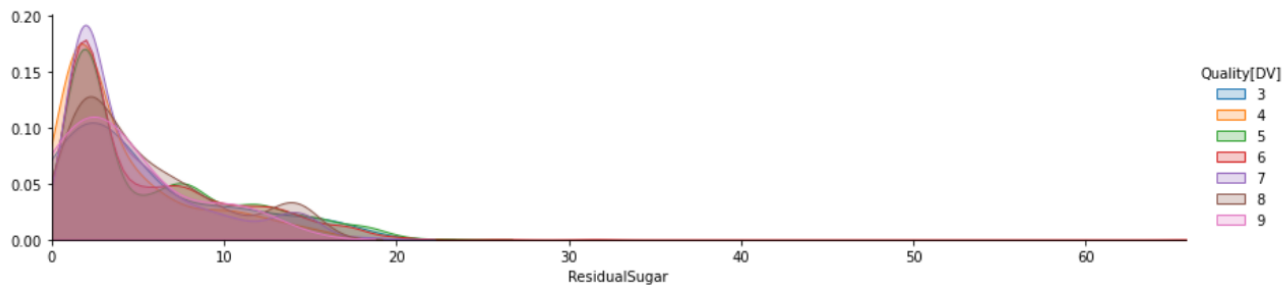
<seaborn.axisgrid.FacetGrid at 0x1c4536ba160>



Column – ResidualSugar:

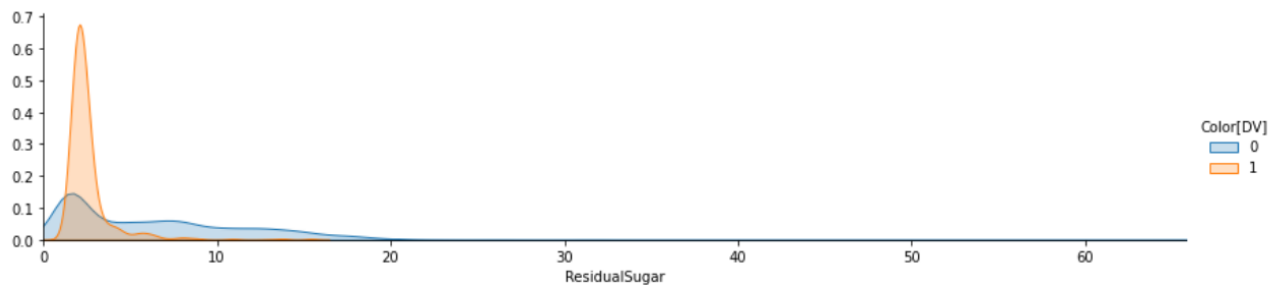
```
#ResidualSugar
facet = sns.FacetGrid(dataset, hue="Quality[DV]", aspect=4)
facet.map(sns.kdeplot, 'ResidualSugar', shade= True)
facet.set(xlim=(0, dataset['ResidualSugar'].max()))
facet.add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x1c453752a00>



```
#ResidualSugar
facet = sns.FacetGrid(dataset, hue="Color[DV]", aspect=4)
facet.map(sns.kdeplot, 'ResidualSugar', shade= True)
facet.set(xlim=(0, dataset['ResidualSugar'].max()))
facet.add_legend()
```

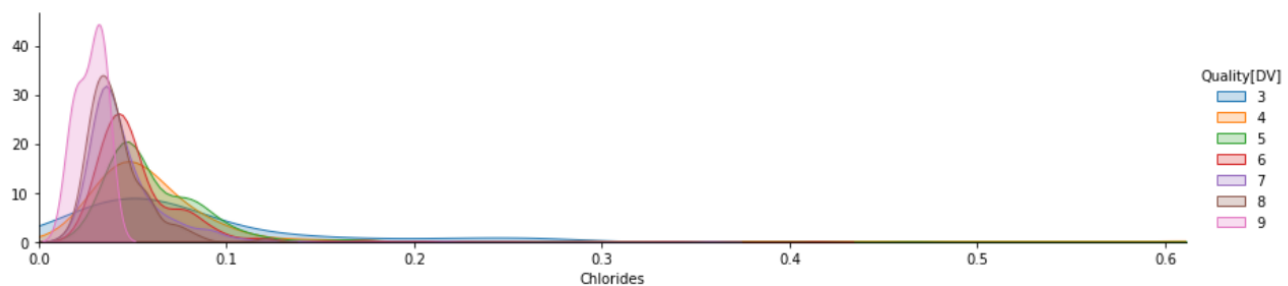
<seaborn.axisgrid.FacetGrid at 0x1c453711af0>



Column – Chlorides:

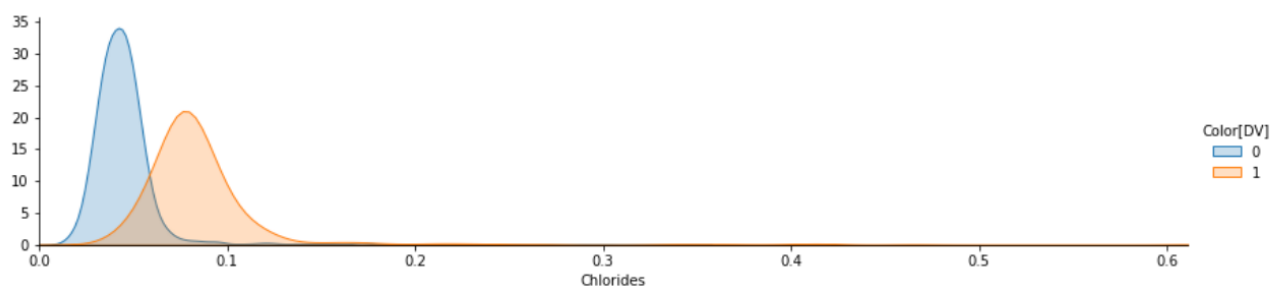
```
#Chlorides
facet = sns.FacetGrid(dataset, hue="Quality[DV]", aspect=4)
facet.map(sns.kdeplot, 'Chlorides', shade= True)
facet.set(xlim=(0, dataset['Chlorides'].max()))
facet.add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x1c4537baee0>



```
#Chlorides
facet = sns.FacetGrid(dataset, hue="Color[DV]", aspect=4)
facet.map(sns.kdeplot, 'Chlorides', shade= True)
facet.set(xlim=(0, dataset['Chlorides'].max()))
facet.add_legend()
```

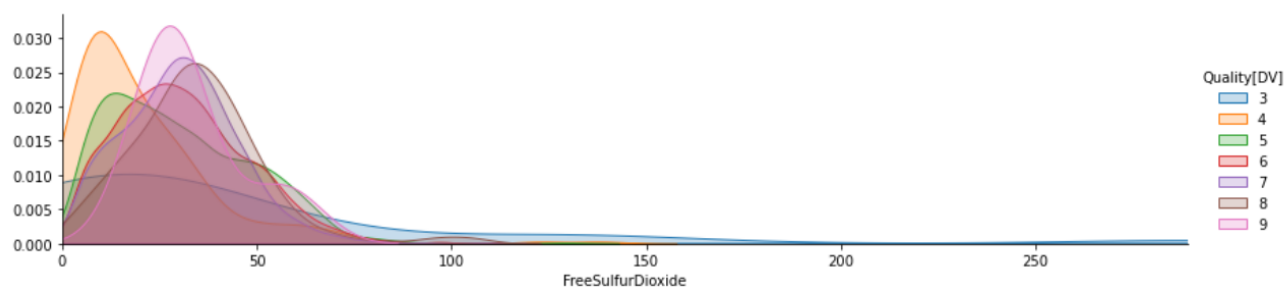
<seaborn.axisgrid.FacetGrid at 0x1c44f565580>



Column – FreeSulfurDioxide:

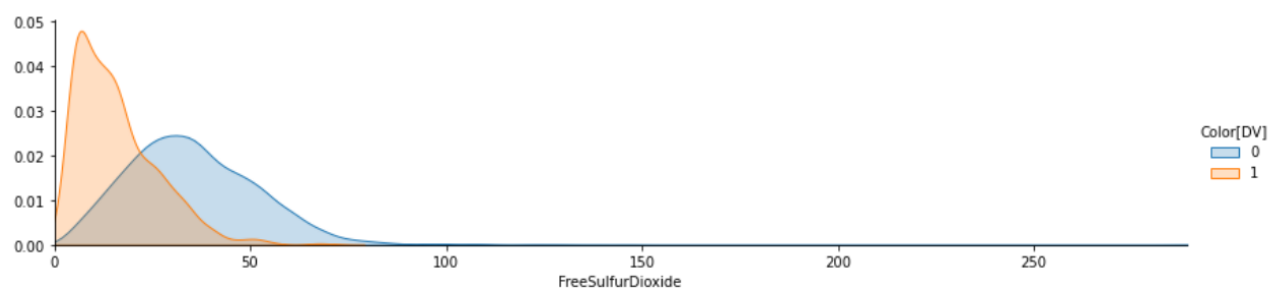
```
#FreeSulfurDioxide
facet = sns.FacetGrid(dataset, hue="Quality[DV]", aspect=4)
facet.map(sns.kdeplot, 'FreeSulfurDioxide', shade=True)
facet.set(xlim=(0, dataset['FreeSulfurDioxide'].max()))
facet.add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x1c453854280>



```
#FreeSulfurDioxide
facet = sns.FacetGrid(dataset, hue="Color[DV]", aspect=4)
facet.map(sns.kdeplot, 'FreeSulfurDioxide', shade=True)
facet.set(xlim=(0, dataset['FreeSulfurDioxide'].max()))
facet.add_legend()
```

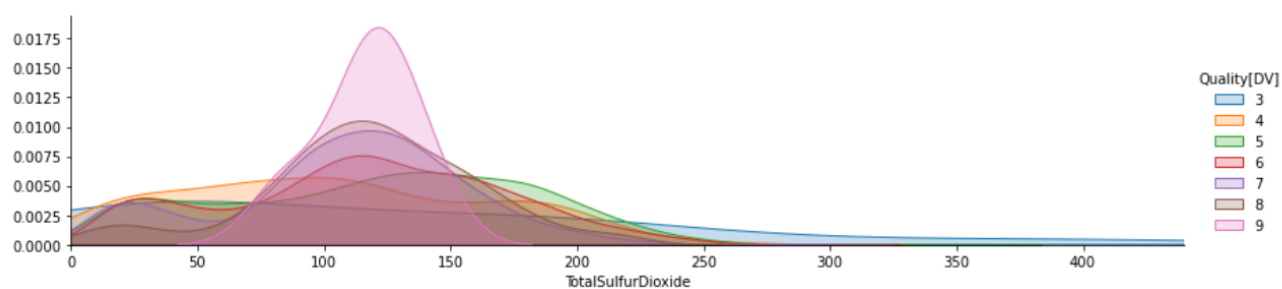
<seaborn.axisgrid.FacetGrid at 0x1c45398f550>



Column – TotalSulfurDioxide:

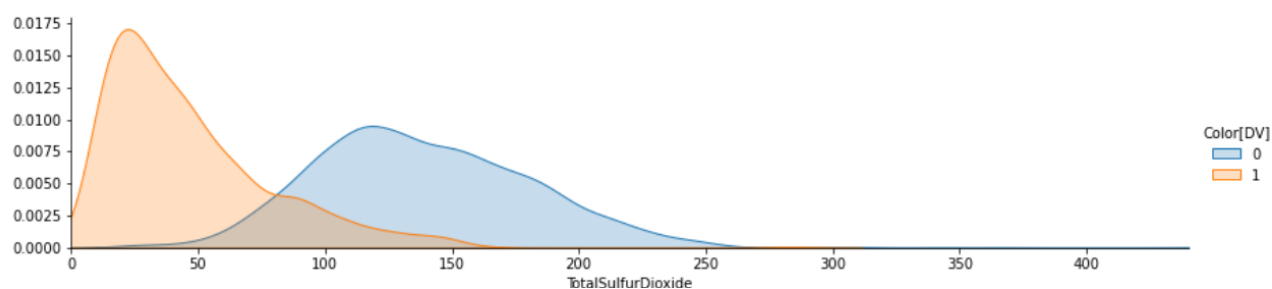
```
#TotalSulfurDioxide
facet = sns.FacetGrid(dataset, hue="Quality[DV]", aspect=4)
facet.map(sns.kdeplot, 'TotalSulfurDioxide', shade=True)
facet.set(xlim=(0, dataset['TotalSulfurDioxide'].max()))
facet.add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x1c453562f40>



```
#TotalSulfurDioxide
facet = sns.FacetGrid(dataset, hue="Color[DV]", aspect=4)
facet.map(sns.kdeplot, 'TotalSulfurDioxide', shade=True)
facet.set(xlim=(0, dataset['TotalSulfurDioxide'].max()))
facet.add_legend()
```

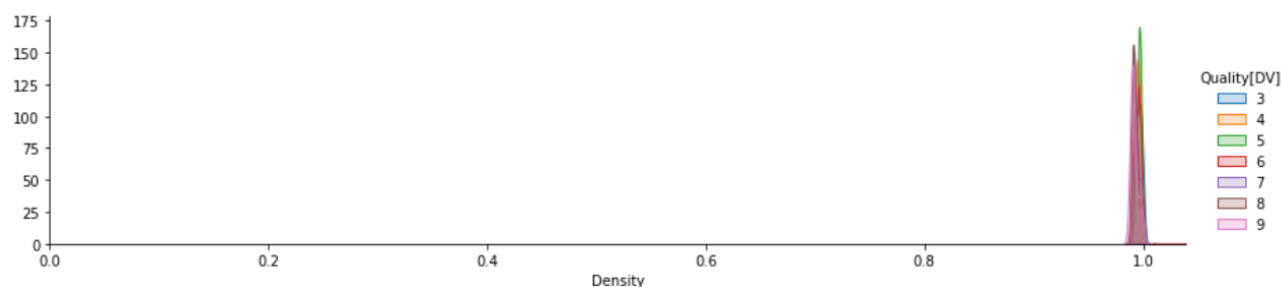
<seaborn.axisgrid.FacetGrid at 0x1c4539a2dc0>



Column – Density:

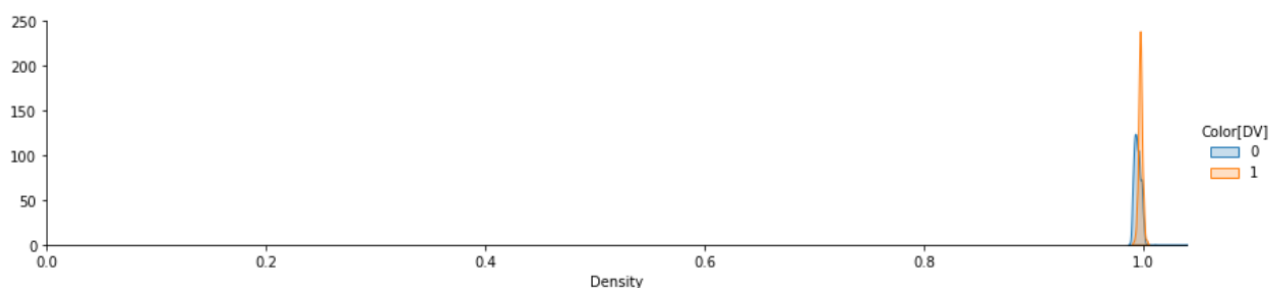
```
#Density
facet = sns.FacetGrid(dataset, hue="Quality[DV]", aspect=4)
facet.map(sns.kdeplot, 'Density', shade= True)
facet.set(xlim=(0, dataset['Density'].max()))
facet.add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x1c455e45490>



```
#Density
facet = sns.FacetGrid(dataset, hue="Color[DV]", aspect=4)
facet.map(sns.kdeplot, 'Density', shade= True)
facet.set(xlim=(0, dataset['Density'].max()))
facet.add_legend()
```

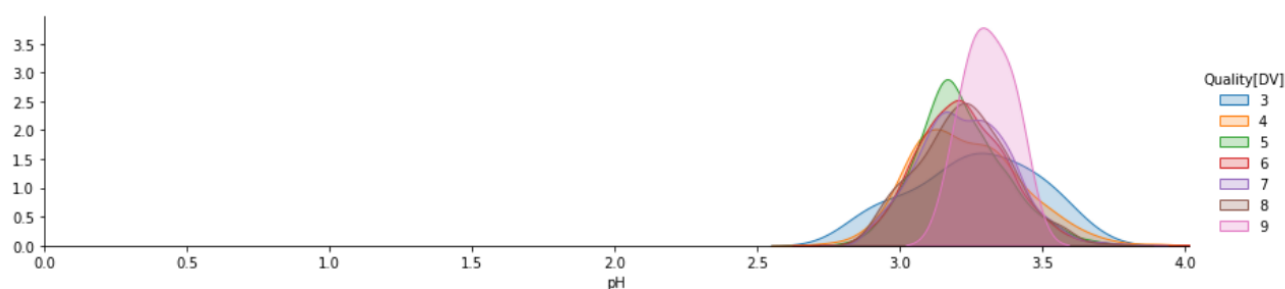
<seaborn.axisgrid.FacetGrid at 0x1c4565e8ee0>



Column – pH:

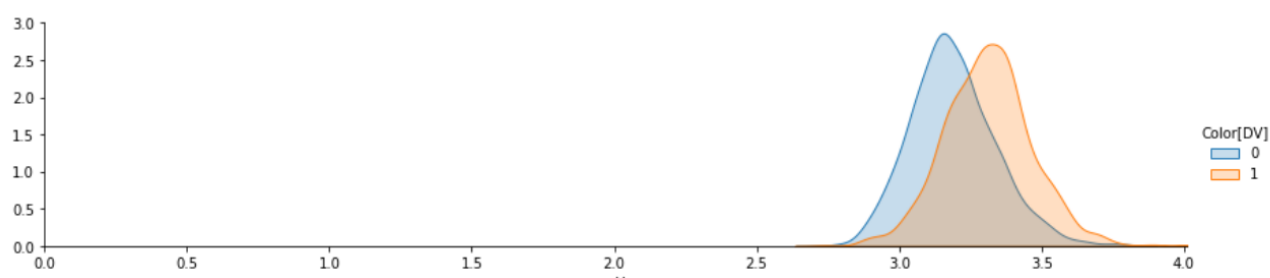
```
#pH
facet = sns.FacetGrid(dataset, hue="Quality[DV]", aspect=4)
facet.map(sns.kdeplot, 'pH', shade= True)
facet.set(xlim=(0, dataset['pH'].max()))
facet.add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x1c455bc1250>



```
#pH
facet = sns.FacetGrid(dataset, hue="Color[DV]", aspect=4)
facet.map(sns.kdeplot, 'pH', shade= True)
facet.set(xlim=(0, dataset['pH'].max()))
facet.add_legend()
```

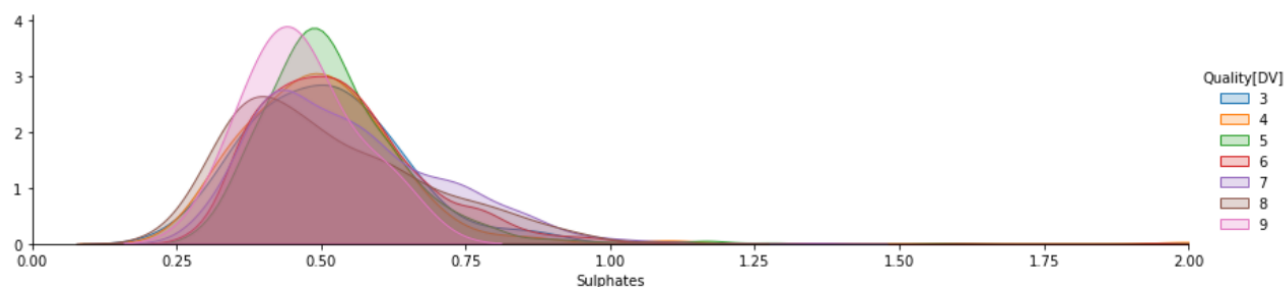
<seaborn.axisgrid.FacetGrid at 0x1c455c4e910>



Column – Sulphates:

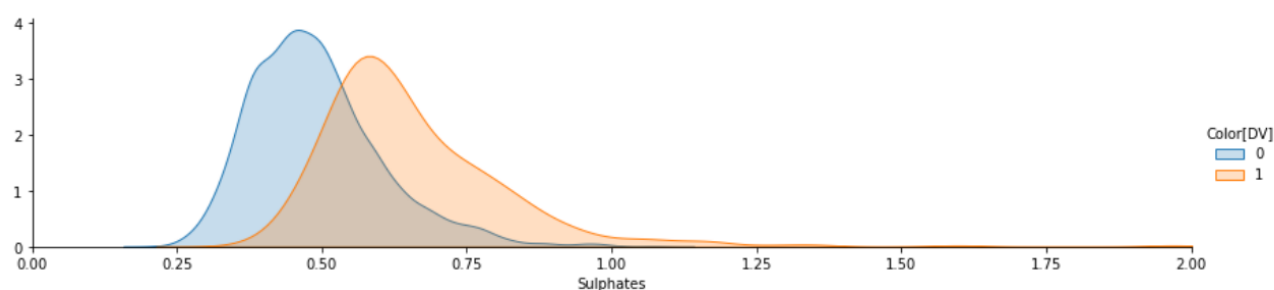
```
#Sulphates
facet = sns.FacetGrid(dataset, hue="Quality[DV]", aspect=4)
facet.map(sns.kdeplot, 'Sulphates', shade= True)
facet.set(xlim=(0, dataset['Sulphates'].max()))
facet.add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x1c455b86eb0>



```
#Sulphates
facet = sns.FacetGrid(dataset, hue="Color[DV]", aspect=4)
facet.map(sns.kdeplot, 'Sulphates', shade= True)
facet.set(xlim=(0, dataset['Sulphates'].max()))
facet.add_legend()
```

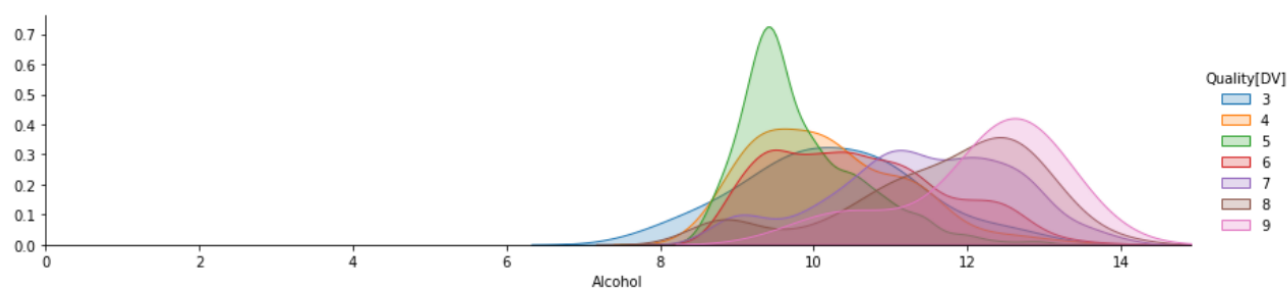
<seaborn.axisgrid.FacetGrid at 0x1c454a8ffa0>



Column – Alcohol:

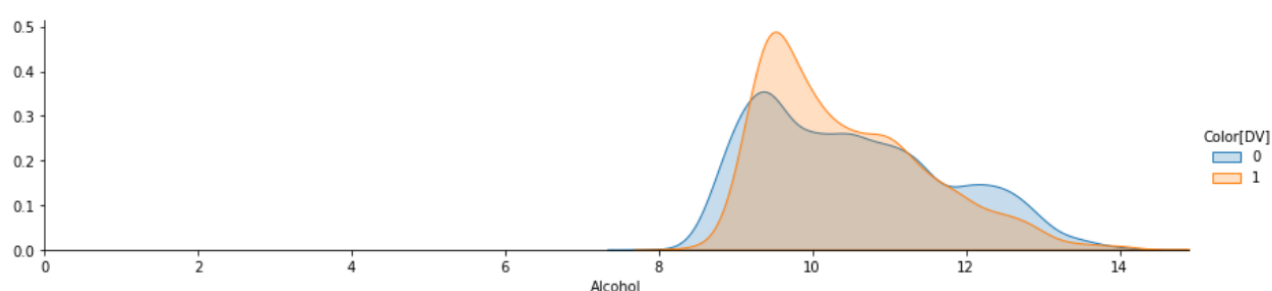
```
#Alcohol
facet = sns.FacetGrid(dataset, hue="Quality[DV]", aspect=4)
facet.map(sns.kdeplot, 'Alcohol', shade= True)
facet.set(xlim=(0, dataset['Alcohol'].max()))
facet.add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x1c455eb65b0>



```
#Alcohol
facet = sns.FacetGrid(dataset, hue="Color[DV]", aspect=4)
facet.map(sns.kdeplot, 'Alcohol', shade= True)
facet.set(xlim=(0, dataset['Alcohol'].max()))
facet.add_legend()
```

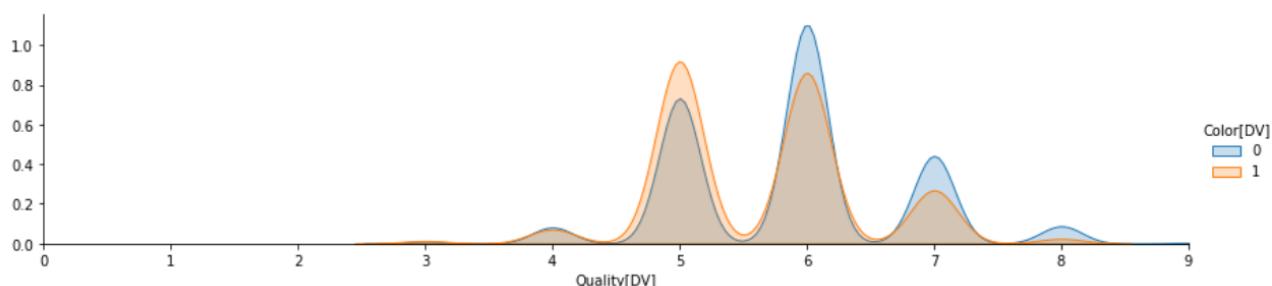
<seaborn.axisgrid.FacetGrid at 0x1c455f5cfa0>



Column – Quality[DV]:

```
#Quality[DV]
facet = sns.FacetGrid(dataset, hue="Color[DV]", aspect=4)
facet.map(sns.kdeplot, 'Quality[DV]', shade= True)
facet.set(xlim=(0, dataset['Quality[DV]'].max()))
facet.add_legend()
```

<seaborn.axisgrid.FacetGrid at 0x1c454aa8a90>



Column – Color[DV]:

We know that the data type of the last column "color" is an object, so we need to convert it into a numeric data type to run the regression model.

```
#Observe the target variable field
dataset['Color[DV]'].value_counts()
```

```
White    4898
Red       1599
Name: Color[DV], dtype: int64
```

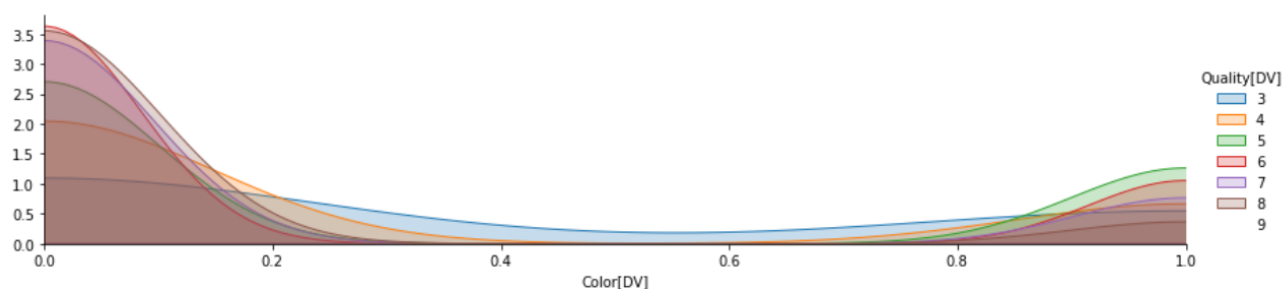
```
#mapping Color
Color_mapping={"White":0,"Red":1}
dataset['Color[DV]'] = dataset['Color[DV]'].map(Color_mapping)
dataset.head(5)#Observe the current dataframe
```

FixedAcidity	VolatileAcidity	CitricAcid	ResidualSugar	Chlorides	FreeSulfurDioxide	TotalSulfurDioxide	Density	pH	Sulphates	Alcohol	Quality[DV]	Color[DV]
6.2	0.270	0.32	6.3	0.048	47.0	159.0	0.99282	3.21	0.60	11.0	6	0
7.4	0.635	0.10	2.4	0.080	16.0	33.0	0.99736	3.58	0.69	10.8	7	1
6.3	0.230	0.33	6.9	0.052	23.0	118.0	0.99380	3.23	0.46	10.4	6	0
6.2	0.630	0.31	1.7	0.088	15.0	64.0	0.99690	3.46	0.79	9.3	5	1
7.6	0.270	0.52	3.2	0.043	28.0	152.0	0.99129	3.02	0.53	11.4	6	0

```
#Color[DV]
facet = sns.FacetGrid(dataset, hue="Quality[DV]", aspect=4)
facet.map(sns.kdeplot, 'Color[DV]', shade= True)
facet.set(xlim=(0, dataset['Color[DV]'].max()))
facet.add_legend()
```

C:\Users\88697\anaconda3\lib\site-packages\seaborn\distributions.py:306: UserWarning: Dataset has 0 variance; skipping density estimate.
warnings.warn(msg, UserWarning)

<seaborn.axisgrid.FacetGrid at 0x1c45618c760>

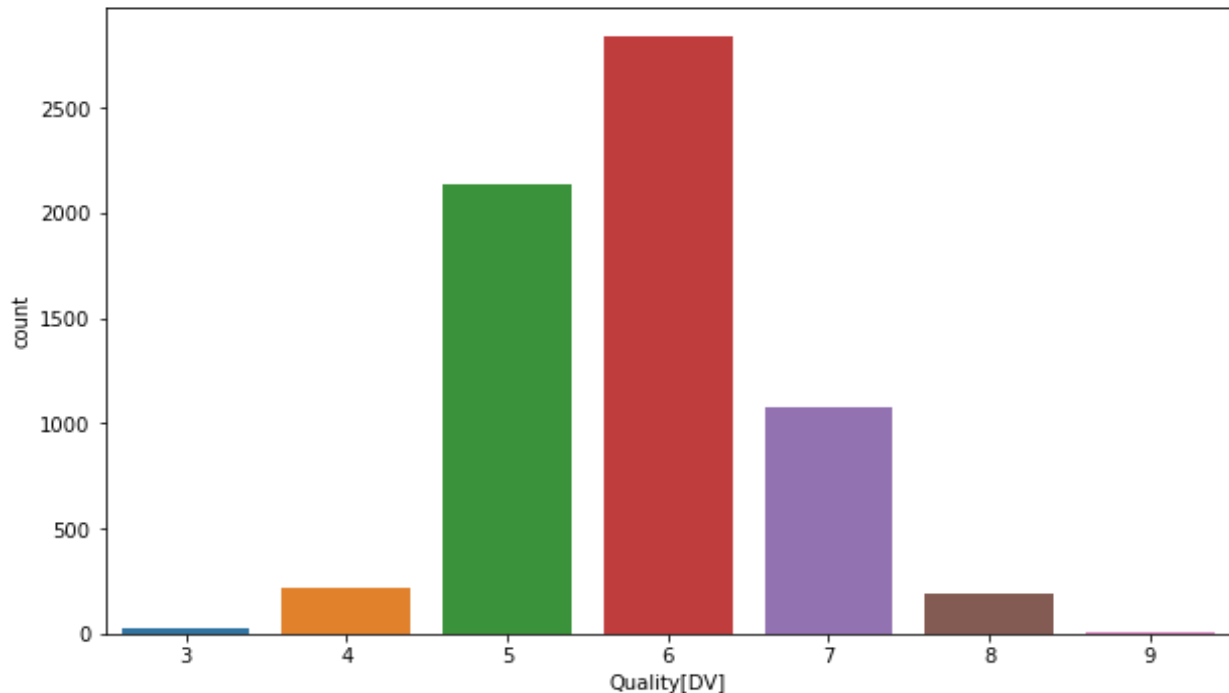


From the above results, we can know that only a few columns have a normal distribution, so we consider all the values to be standardized before training the model.

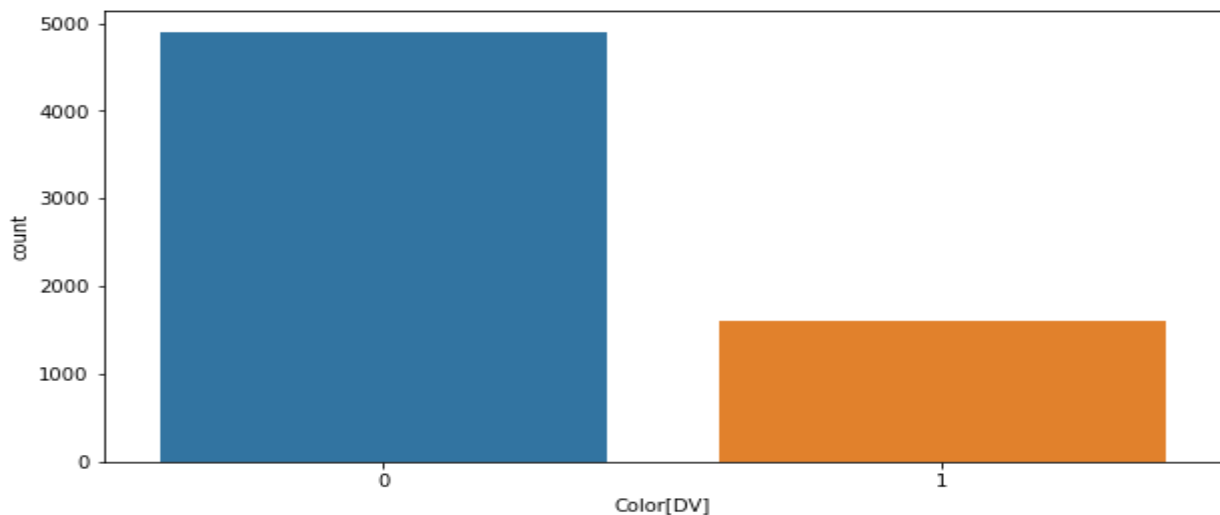
Step 2: - Check the entire data set:

We can use countplot to help us check whether the data set is balanced. We use countplot to calculate how many records there are for each value of "quality" and calculate how many records there are for each value of "color". This will help us understand whether the data set is balanced.

```
sns.countplot(dataset['Quality[DV]'])
```



```
sns.countplot(dataset['Color[DV]'])
```



From the above picture, we can know that the data set is unbalanced. If the data set is unbalanced, we can use SMOTE to solve these problems.

We can try all the above methods. We can train the model without pre-processing the data, or train the model with standardized data, or can also train the model after processing the data with the SMOTE, we can also use cross-validation. Or a combination of the above processing: standardized the data with the SMOTE and add cross-validation then train the model. Finally, compare the performance of the models after these different data processing methods.

Modeling

After the data is well-prepared, we can train the model, as long as the model is trained, we can make predictions. For the regression, we will train LinearRegression, Support Vector Regression, SGDRegressor, KNeighborsRegressor, DecisionTreeRegressor, RandomForestRegressor, GradientBoostingRegressor, and MLPRegressor.

Without data pre-processing:

Before comparing the model performance of each data pre-processing method, we don't know which data pre-processing method is most suitable for this data. So, we also need the model results without data pre-processing.

Before we run the model, we need to split the dataset for regression:

```
X = dataset.drop('Quality[DV]',axis=1)
Y = dataset['Quality[DV]']
X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, test_size = 0.3, random_state = 100)
```

The following results are the results of running the model without any pre-processing.

LinearRegression:

```
from sklearn import linear_model
reg=linear_model.LinearRegression()
reg.fit(X_train, Y_train)
```

```
LinearRegression()
```

Support Vector Regression:

```
from sklearn import svm
SVR = svm.SVR(kernel='linear')
SVR.fit(X_train, Y_train)
```

```
SVR(kernel='linear')
```

(Stochastic Gradient Descent)SGDRegressor

```
from sklearn.linear_model import SGDRegressor
SGDr = SGDRegressor(penalty='elasticnet', alpha=0.001, max_iter=1100, random_state=100, learning_rate='
#loss={'squared_error', 'huber', 'epsilon_insensitive', 'squared_epsilon_insensitive'}
#penalty={'l2', 'l1', 'elasticnet'}
#learning_rate={'constant', 'optimal', 'invscaling', 'adaptive'})
SGDr.fit(X_train, Y_train)
```

```
SGDRegressor(alpha=0.001, learning_rate='optimal', max_iter=1100,
              penalty='elasticnet', random_state=100)
```

KNeighborsRegressor:

```
from sklearn.neighbors import KNeighborsRegressor
KNNr = KNeighborsRegressor(n_neighbors=7, p=1)
KNNr.fit(X_train, Y_train)
```

```
KNeighborsRegressor(n_neighbors=7, p=1)
```

DecisionTreeRegressor:

```
from sklearn.tree import DecisionTreeRegressor
DTr = DecisionTreeRegressor(criterion='friedman_mse',max_depth=9,max_features='auto')
#criterion={"squared_error", "friedman_mse", "absolute_error", "poisson"}
#max_features=int, float or {"auto", "sqrt", "log2"}
DTr.fit(X_train, Y_train)
```

```
DecisionTreeRegressor(criterion='friedman_mse', max_depth=9,
                      max_features='auto')
```

RandomForestRegressor:

```
from sklearn.ensemble import RandomForestRegressor
RFR = RandomForestRegressor(criterion='absolute_error', max_depth=45, max_features='sqrt', random_state=100)
# criterion = {"squared_error", "absolute_error", "poisson"}
# max_features = int, float or {"auto", "sqrt", "log2"}
RFR.fit(X_train, Y_train)
```

RandomForestRegressor(criterion='absolute_error', max_depth=45, max_features='sqrt', random_state=100)

GradientBoostingRegressor

```
from sklearn.ensemble import GradientBoostingRegressor
GBR = GradientBoostingRegressor(loss='absolute_error', min_samples_leaf=2, max_depth=9, max_features='sqrt', random_state=100)
# loss = {"squared_error", "absolute_error", "huber", "quantile"}
# criterion = {"friedman_mse", "squared_error", "mse", "mae"}
# max_features = {"auto", "sqrt", "log2"}
GBR.fit(X_train, Y_train)
```

GradientBoostingRegressor(loss='absolute_error', max_depth=9, max_features='sqrt', min_samples_leaf=2, random_state=100)

MLPRegressor:

```
from sklearn.neural_network import MLPRegressor
MLP = MLPRegressor(activation='tanh', batch_size=125, random_state=1, max_iter=1500)
# (hidden_layer_sizes=(100))
# activation = {"identity", "logistic", "tanh", "relu"}
# solver = {"lbfgs", "sgd", "adam"}
# learning_rate = {"constant", "invscaling", "adaptive"}
MLP.fit(X_train, Y_train)
```

MLPRegressor(activation='tanh', batch_size=125, max_iter=1500, random_state=1)

For the classification, we will run LogisticRegression, Support Vector Classification, SGDClassifier, KNeighborsClassifier, DecisionTreeClassifier, RandomForestClassifier, AdaBoostClassifier, and MLPClassifier.

Before we run the model, we need to split the dataset for classification:

```
X = dataset[dataset.columns[:12]]
Y = dataset['Color[DV]']
X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, test_size = 0.3, random_state = 100)
```

LogisticRegression:

```
from sklearn.linear_model import LogisticRegression
Logistic = LogisticRegression(max_iter=700)
# penalty = {"l1", "l2", "elasticnet", "none"}
# solver = {"newton-cg", "lbfgs", "liblinear", "sag", "saga"}
# multi_class = {"auto", "ovr", "multinomial"}
Logistic.fit(X_train, Y_train)
```

LogisticRegression(max_iter=700)

C-Support Vector Classification:

```
from sklearn import svm
SVC = svm.SVC(kernel='linear')
# kernel = {"linear", "poly", "rbf", "sigmoid", "precomputed"}
# decision_function_shape = {"ovo", "ovr"}
SVC.fit(X_train, Y_train)
```

SVC(kernel='linear')

SGDClassifier:

```
from sklearn.linear_model import SGDClassifier
SDGc = SGDClassifier(loss='perceptron',penalty='elasticnet',alpha=0.0009, random_state=100)
#loss={'hinge', 'log', 'modified_huber', 'squared_hinge', 'perceptron'}
#penalty={'l2', 'l1', 'elasticnet'}
#Learning_rate={'constant', 'optimal', 'invscaling', 'adaptive'}
SDGc.fit(X_train, Y_train)
```

```
SGDClassifier(alpha=0.0009, loss='perceptron', penalty='elasticnet',
              random_state=100)
```

KNeighborsClassifier:

```
from sklearn.neighbors import KNeighborsClassifier
KNNc = KNeighborsClassifier(n_neighbors=2,p = 1)
KNNc.fit(X_train, Y_train)
```

```
KNeighborsClassifier(n_neighbors=2, p=1)
```

DecisionTreeClassifier:

```
from sklearn.tree import DecisionTreeClassifier
DTc = DecisionTreeClassifier(criterion='entropy',max_depth=8,min_samples_split=3,random_state=0)
#criterion={"gini", "entropy"}
#max_features=int, float or {"auto", "sqrt", "log2"}
DTc.fit(X_train, Y_train)
```

```
DecisionTreeClassifier(criterion='entropy', max_depth=8, min_samples_split=3,
                      random_state=0)
```

RandomForestClassifier:

```
from sklearn.ensemble import RandomForestClassifier
RFC = RandomForestClassifier(n_estimators=250,criterion='entropy',max_features='auto',max_depth=11,min_samples_split=3)
#criterion={"gini", "entropy"}
#max_features=int, float or {"auto", "sqrt", "log2"}
RFC.fit(X_train, Y_train)
```

```
RandomForestClassifier(criterion='entropy', max_depth=11, min_samples_split=3,
                      n_estimators=250)
```

AdaBoostClassifier:

```
from sklearn.ensemble import AdaBoostClassifier
ABc = AdaBoostClassifier()
#algorithm={'SAMME', 'SAMME.R'}
ABc.fit(X_train, Y_train)
```

```
AdaBoostClassifier()
```

MLPRegressor:

```
from sklearn.neural_network import MLPClassifier
MLPc = MLPClassifier(activation='tanh',batch_size=100,random_state=100,max_iter=1000)
#(hidden_layer_sizes=(100))
#activation={'identity', 'logistic', 'tanh', 'relu'}
#solver={'lbfgs', 'sgd', 'adam'}
#learning_rate={'constant', 'invscaling', 'adaptive'}
MLPc.fit(X_train, Y_train)
```

```
MLPClassifier(activation='tanh', batch_size=100, max_iter=1000,
              random_state=100)
```

For the complete code, please refer to the APPENDIX-1

Not only running ordinary models, but also ensemble models (like RandomForest etc.) and network neural models. Since we don't know which model is most suitable for this set of data, the more models we try, the higher the probability of finding the best model. For each model, I try and error to find the best parameters, because I tried to find out the best performance of the model.

(Improve the model)

Standardize the data:

From the “check every column” part, we can know that many column data do not meet the normal distribution, we can consider normalization or standardization. We use standardization to make the data meet the normal distribution. We can observe the performance of the model after standardization. The standardization we use here is Z-Score Normalization, which is not normalization, because normalization does not change the distribution pattern of data. However, changing the data distribution may affect the prediction of the model.

```
X = dataset.drop('Quality[DV]',axis=1)
Y = dataset['Quality[DV]']

from sklearn import preprocessing
z_score_scaler = preprocessing.StandardScaler()
X_standardized = z_score_scaler.fit_transform(X)

X_train, X_test, Y_train, Y_test = train_test_split(
    X_standardized, Y, test_size = 0.3, random_state = 100)
```

For the complete code, please refer to the APPENDIX-2

Use cross-validation:

Using cross-validation we can reduce the chance of overfitting, which can make the model perform better. The cross-validation we use here is k-fold cross-validation. The advantage of this method is that it repeatedly uses randomly generated sub-samples for training and validation. 10 cross-validation is the most common. Therefore, we use 10 cross-validation. But the model training time will be quite longer that sometimes the model cannot be trained on a personal computer.

```
X = dataset.drop('Quality[DV]',axis=1)
Y = dataset['Quality[DV]']

from sklearn.model_selection import KFold
kfold = KFold(n_splits=10,shuffle=True)

#fold.split會將dataset拆成train,test
for train, test in kfold.split(dataset):
    X_train =X.iloc[train]
    Y_train =Y.iloc[train]
    X_test =X.iloc[test]
    Y_test =Y.iloc[test]
```

For the complete code, please refer to the APPENDIX-3

Use SMOTE(Synthetic Minority Oversampling Technique):

From the “check the entire data set, we can know that the dataset is unbalanced. In this case, we can use SOMTE. SMOTE is a well-known sample generation method, after the data is generated, we can solve the problem of dataset unbalanced. But the problem of using this is that the algorithm may distort the importance of features.

```
X = dataset.drop('Quality[DV]',axis=1)
Y = dataset['Quality[DV]']

X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, test_size = 0.3, random_state = 100)

from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=42,k_neighbors=4)
X_SMOTE,Y_SMOTE = smote.fit_resample(X_train, Y_train)
```

For the complete code, please refer to the APPENDIX-4

After using the above methods alone, we can also combine the above methods to pre-process the data. As mentioned earlier, we will not know what kind of processing method is most suitable for this dataset before the models are compared. Perhaps the model can perform better after combining the data processing method.

Standardize data and use cross-validation:

The following data pre-processing method combined data standardization with cross-validation. The Z-score is an array, we need to convert it into a data frame so that we can use K-fold cross-validation.

```
X = dataset.drop('Quality[DV]',axis=1)
Y = dataset['Quality[DV]']

from sklearn import preprocessing
z_score_scaler = preprocessing.StandardScaler()
X_z_score = z_score_scaler.fit_transform(X)

#iloc只能用在df上，而Z_score是array所以要array轉df
X=pd.DataFrame(X_z_score)

from sklearn.model_selection import KFold
kfold = KFold(n_splits=10,shuffle=True)
#fold.split會將dataset拆成train,test
for train, test in kfold.split(dataset):
    X_train =X.iloc[train]
    Y_train =Y.iloc[train]
    X_test =X.iloc[test]
    Y_test =Y.iloc[test]
```

For the complete code, please refer to the APPENDIX-5

Standardize data and use SMOTE:

The following data pre-processing method combined data standardization with SMOTE.

```
X = dataset.drop('Quality[DV]',axis=1)
Y = dataset['Quality[DV]']

from sklearn import preprocessing
z_score_scaler = preprocessing.StandardScaler()
X_z_score = z_score_scaler.fit_transform(X)

X_train, X_test, Y_train, Y_test = train_test_split(
    X_z_score, Y, test_size = 0.3, random_state = 100)

from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=42,k_neighbors=4)
X_SMOTE,Y_SMOTE = smote.fit_resample(X_train, Y_train)
```

For the complete code, please refer to the APPENDIX-6

Use cross-validation and use SMOTE:

The following data pre-processing methods combined cross-validation with SMOTE.

```
X = dataset.drop('Quality[DV]',axis=1)
Y = dataset['Quality[DV]']

from sklearn.model_selection import KFold
kfold = KFold(n_splits=10,shuffle=True)
#fold.split會將dataset拆成train,test
for train, test in kfold.split(dataset):
    X_train =X.iloc[train]
    Y_train =Y.iloc[train]
    X_test =X.iloc[test]
    Y_test =Y.iloc[test]

from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=42,k_neighbors=3)
X_SMOTE,Y_SMOTE = smote.fit_resample(X_train, Y_train)
```

For the complete code, please refer to the APPENDIX-7

Standardize data, use cross-validation and use SMOTE:

Finally, all the data pre-processing methods are used in this data set at the same time.

```
X = dataset.drop('Quality[DV]',axis=1)
Y = dataset['Quality[DV]']

from sklearn import preprocessing
z_score_scaler = preprocessing.StandardScaler()
X_z_score = z_score_scaler.fit_transform(X)

#iloc只能在df上，而Z_score是array所以要array轉df
X=pd.DataFrame(X_z_score)

from sklearn.model_selection import KFold
kfold = KFold(n_splits=10,shuffle=True)
#fold.split會將dataset拆成train,test
for train, test in kfold.split(dataset):
    X_train =X.iloc[train]
    Y_train =Y.iloc[train]
    X_test =X.iloc[test]
    Y_test =Y.iloc[test]

from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=42,k_neighbors=4)
X_SMOTE,Y_SMOTE = smote.fit_resample(X_train, Y_train)
```

For the complete code, please refer to the APPENDIX-7

To sum up, we have eight ways to deal with the dataset in total. We run the models mentioned earlier and adjust the parameters one by one to make the performance of the model be better.

Take SGDRegressor as an example, as you can see below, after eight different data pre-processing methods, the parameter settings are different each time.

```
from sklearn.linear_model import SGDRegressor
SGDr = SGDRegressor(penalty='elasticnet', alpha=0.001, max_iter=1100, random_state=100, lea
#Loss={'squared_error', 'huber', 'epsilon_insensitive', 'squared_epsilon_insensitive'}
#penalty={'l2', 'l1', 'elasticnet'}
#Learning_rate={'constant', 'optimal', 'invscaling', 'adaptive'}
SGDr.fit(X_train, Y_train)
```

Out[44]:

```
SGDRegressor(alpha=0.001, learning_rate='optimal', max_iter=1100,
              penalty='elasticnet', random_state=100)
```

```
from sklearn.linear_model import SGDRegressor
SGDr = SGDRegressor(loss='squared_error',penalty='elasticnet', alpha=0.001,learning_rate='a
#Loss={'squared_error', 'huber', 'epsilon_insensitive', 'squared_epsilon_insensitive'}
#penalty={'l2', 'l1', 'elasticnet'}
#Learning_rate={'constant', 'optimal', 'invscaling', 'adaptive'}
SGDr.fit(X_train, Y_train)
```

Out[13]:

```
SGDRegressor(alpha=0.001, learning_rate='adaptive', penalty='elasticnet')
```

```
from sklearn.linear_model import SGDRegressor
SGDr = SGDRegressor(loss='epsilon_insensitive',penalty='elasticnet', alpha=0.0008, random_s
#Loss={'squared_error', 'huber', 'epsilon_insensitive', 'squared_epsilon_insensitive'}
#penalty={'l2', 'l1', 'elasticnet'}
#Learning_rate={'constant', 'optimal', 'invscaling', 'adaptive'}
SGDr.fit(X_train, Y_train)
```

Out[13]:

```
SGDRegressor(alpha=0.0008, loss='epsilon_insensitive', penalty='elasticnet',
              random_state=100)
```

```

from sklearn.linear_model import SGDRegressor
SGDr=SGDRegressor(loss='epsilon_insensitive',penalty='l1',learning_rate='adaptive',random_s
#loss={'squared_error','huber','epsilon_insensitive','squared_epsilon_insensitive'}
#penalty={'l2','l1','elasticnet'}
#learning_rate={'constant','optimal','invscaling','adaptive'}
SGDr.fit(X_SMOTE, Y_SMOTE)

```

Out[9]:

```

SGDRegressor(learning_rate='adaptive', loss='epsilon_insensitive', penalty='l
1',
              random_state=10)

```

```

from sklearn.linear_model import SGDRegressor
SGDr = SGDRegressor(loss='epsilon_insensitive',penalty='elasticnet',learning_rate='adaptive
#loss={'squared_error','huber','epsilon_insensitive','squared_epsilon_insensitive'}
#penalty={'l2','l1','elasticnet'}
#learning_rate={'constant','optimal','invscaling','adaptive'}
SGDr.fit(X_train, Y_train)

```

Out[13]:

```

SGDRegressor(learning_rate='adaptive', loss='epsilon_insensitive',
              penalty='elasticnet')

```

```

from sklearn.linear_model import SGDRegressor
SGDr = SGDRegressor(loss='huber',penalty='l1',learning_rate='adaptive',alpha=0.5)
#loss={'squared_error','huber','epsilon_insensitive','squared_epsilon_insensitive'}
#penalty={'l2','l1','elasticnet'}
#learning_rate={'constant','optimal','invscaling','adaptive'}
SGDr.fit(X_SMOTE, Y_SMOTE)

```

Out[9]:

```

SGDRegressor(alpha=0.5, learning_rate='adaptive', loss='huber', penalty='l1')

```

```

from sklearn.linear_model import SGDRegressor
SGDr = SGDRegressor(loss='squared_epsilon_insensitive',learning_rate='constant',max_iter=10
#loss={'squared_error','huber','epsilon_insensitive','squared_epsilon_insensitive'}
#penalty={'l2','l1','elasticnet'}
#learning_rate={'constant','optimal','invscaling','adaptive'}
SGDr.fit(X_SMOTE, Y_SMOTE)

```

Out[13]:

```

SGDRegressor(alpha=0.006, learning_rate='constant',
              loss='squared_epsilon_insensitive', random_state=100)

```

```

from sklearn.linear_model import SGDRegressor
SGDr = SGDRegressor(penalty='l1',learning_rate='optimal',alpha=0.001,random_state=100)
#loss={'squared_error','huber','epsilon_insensitive','squared_epsilon_insensitive'}
#penalty={'l2','l1','elasticnet'}
#learning_rate={'constant','optimal','invscaling','adaptive'}
SGDr.fit(X_SMOTE, Y_SMOTE)

```

Out[13]:

```

SGDRegressor(alpha=0.001, learning_rate='optimal', penalty='l1',
              random_state=100)

```

Evaluation

After training the model, we can indeed make predictions, but we don't know whether the predictions are accurate or not, so we need to evaluate the model to know how the model performs and whether it will be accurate when using the model to make predictions in the future. If the performance is not good, we can try to improve the model to get better model performance results. Finally, compare all the model performance results we have tried before to find the best model.

The performance of the numerical prediction task evaluates the difference between the predicted value and the actual value. There are two representations of error: mean square error (MSE) and mean absolute error (MAE). The mean square error has an amplifying effect (squared) for Outliers when the mean absolute just simply shows the overall error. Both of these values are quite informative.

```
y_predMLP = MLP.predict(X_test)
mse_valid = mean_squared_error(Y_test, y_predMLP)
print(mse_valid)
mae_valid = mean_absolute_error(Y_test, y_predMLP)
print(mae_valid)#log0.766
```

After evaluating each model, we organize all performance values in the following table:

Regression		No process	stand ardiz ed	k- fold	SMOTE	STD & k-fold	SMOTE & kfold	STD & SMOTE	STD SMOTE kfold
LinearRegression	MSE	0.540	0.540	0.517	1.078	0.544	1.031	1.066	1.150
	MAE	0.573	0.573	0.558	0.824	0.576	0.817	0.821	0.794
SVR	MSE	0.545	0.541	0.520		0.549		1.396	1.746
	MAE	0.572	0.570	0.556		0.576		0.935	0.957
SGDRegressor	MSE	1.903	0.540	0.709	1.227	0.549	0.743	0.970	1.183
	MAE	3T	0.573	0.654	0.877	0.577	0.607	0.649	0.797
KNeighborsRegressor	MSE	0.637	0.498	0.592	1.240	0.471	1.145	0.773	0.737
	MAE	0.631	0.540	0.593	0.852	0.526	0.806	0.649	0.651
DecisionTreeRegressor	MSE	0.583	0.539	0.545	0.850	0.510	0.729	0.842	0.745
	MAE	0.557	0.561	0.571	0.694	0.551	0.651	0.749	0.638
RandomForestRegressor	MSE	0.445	0.377	0.310	0.398	0.345	0.671	0.395	0.329
	MAE	0.484	0.441	0.403	0.449	0.423	0.663	0.446	0.408
GradientBoostingRegressor	MSE	0.423	0.372	0.315	0.407	0.344	0.370	0.401	0.374
	MAE	0.467	0.405	0.354	0.420	0.381	0.384	0.419	0.442
MLPRegressor	MSE	0.519	0.471	0.511	0.825	0.470	0.853	0.828	0.815
	MAE	0.570	0.533	0.547	0.657	0.535	0.727	0.700	0.702

The blank part is because of the SMOTE method, the model can't be trained on my computer.

From the above table, we can see that the ensemble models – RandomForestRegressor and GradientBoostingRegressor with k-fold perform best.

The performance of the category prediction task is measured by the number of misclassifications. There are two representations of error: confusion matrix and average accuracy.

There are four measurements that show the performance of the category prediction task, they are True Negative (TN), True Positive (TP), False Negative (FN), and False Positive (FP). These four measurements can be expressed in a matrix called a Confusion matrix. In addition, we can also use accuracy to simply show the performance of category prediction tasks.

```
y_predMLPc = MLPc.predict(X_test)
mse_valid = mean_squared_error(Y_test, y_predMLPc)
print(mse_valid)
mae_valid = mean_absolute_error(Y_test, y_predMLPc)
print(mae_valid)
print("\nConfusion Matrix: \n",confusion_matrix(Y_test, y_predMLPc))
print ("Accuracy : ",accuracy_score(Y_test,y_predMLPc)*100)
```

After evaluating each model, we organize all performance values in the following table:

Classification		No process	standardized	k-fold	SMOTE	STD & k-fold	SMOTE & kfold	STD & SMOTE	STD & SMOTE
LogisticRegression	CM	[...16] [14...]	[...05] [05...]	[...04] [06...]	[...32] [08...]	[...01] [00...]	[...05] [02...]	[...05] [05...]	[...04] [01...]
	Acc	98.46	99.49	98.46	97.95	99.85	98.92	99.49	99.23
SVC	CM	[...13] [12...]	[...03] [05...]	[...01] [04...]	[...21] [07...]	[...01] [00...]	[...07] [02...]	[...00] [05...]	[...04] [01...]
	Acc	98.72	99.59	99.23	98.56	99.85	98.61	99.74	99.23
SGDClassifier	CM	[...35] [53...]	[...04] [07...]	[...15] [10...]	[...43] [48...]	[...01] [01...]	[...21] [12...]	[...04] [04...]	[...05] [01...]
	Acc	95.48	99.44	96.15	95.33	99.69	94.92	99.59	99.08
KNeighborsClassifier	CM	[...21] [60...]	[...01] [08...]	[...06] [18...]	[...47] [37...]	[...01] [02...]	[...13] [12...]	[...05] [04...]	[...05] [01...]
	Acc	95.85	99.54	96.30	95.69	99.54	96.15	99.54	99.08
DecisionTreeClassifier	CM	[...07] [10...]	[...09] [10...]	[...02] [02...]	[...07] [14...]	[...01] [01...]	[...06] [04...]	[...11] [08...]	[...06] [01...]
	Acc	99.13	99.02	99.38	98.92	99.69	98.45	99.02	98.92
RandomForestClassifier	CM	[...00] [06...]	[...01] [05...]	[...00] [01...]	[...01] [05...]	[...00] [00...]	[...01] [02...]	[...01] [05...]	[...03] [01...]
	Acc	99.69	99.69	99.84	99.69	100	99.53	99.69	99.38
AdaBoostClassifier	CM	[...04] [06...]	[...04] [06...]	[...02] [00...]	[...06] [05...]	[...02] [00...]	[...00] [01...]	[...03] [07...]	[...09] [01...]
	Acc	99.48	99.49	99.69	99.44	99.69	99.84	99.48	98.46
MLPClassifier	CM	[...15] [08...]	[...02] [04...]	[...02] [09...]	[...18] [07...]	[...00] [00...]	[...01] [03...]	[...01] [02...]	[...02] [01...]
	Acc	98.82	99.69	99.31	98.71	100	99.38	99.84	99.54

From the above table, we can see that the ensemble model - random forest and neural network model - MLPRegressor with data pre-processing of standardize the data and then use the cross-validation perform the best.

Deployment

Regarding the presentation of results, I think it can be divided into two parts, one is data visualization to show the model results, the other one is to apply the results for practical application.

Visualization:

First of all, take a look at the two best performing models RandomForestRegressor and GradientBoostingRegressor. we can use the attribute "feature_importances_" of Regressor to help us understand how the regression trend is determined. In other words, help us understand how the weight of each column is assigned.

RandomForestRegressor

FEATURE IMPORTANCES	
FixedAcidity	0.062971
VolatileAcidity	0.106314
CitricAcid	0.075028
ResidualSugar	0.073943
Chlorides	0.084506
FreeSulfurDioxide	0.088429
TotalSulfurDioxide	0.077316
Density	0.111112
pH	0.067135
Sulphates	0.076262
Alcohol	0.172733
Color[DV]	0.004250
dtype: float64	

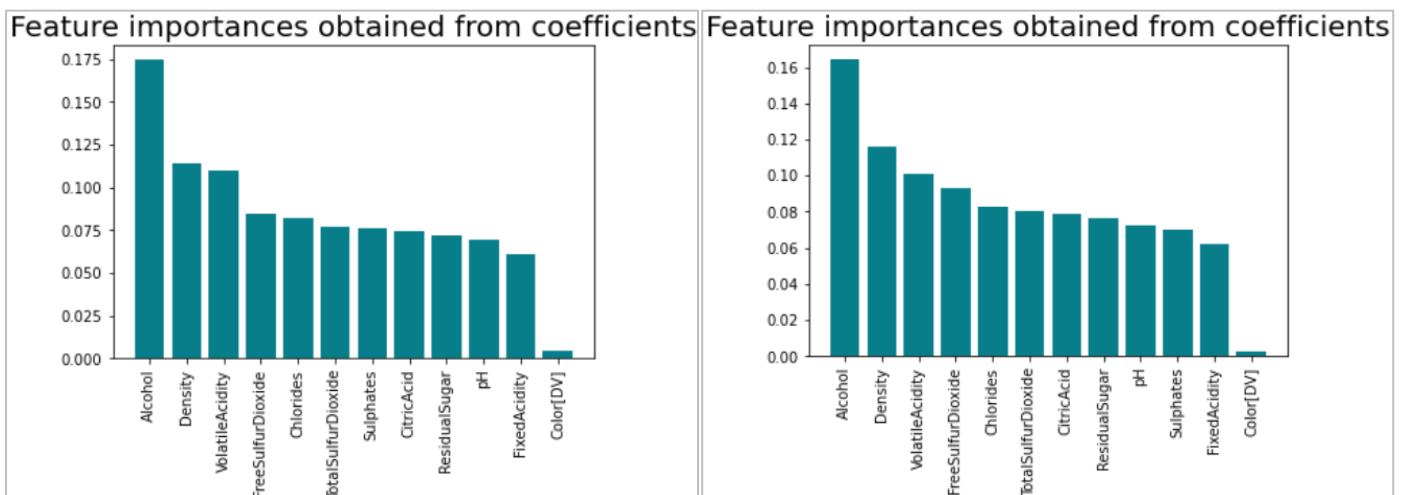
GradientBoostingRegressor

FEATURE IMPORTANCES	
FixedAcidity	0.062303
VolatileAcidity	0.100869
CitricAcid	0.078773
ResidualSugar	0.076466
Chlorides	0.082549
FreeSulfurDioxide	0.092792
TotalSulfurDioxide	0.080609
Density	0.115727
pH	0.072552
Sulphates	0.070203
Alcohol	0.164093
Color[DV]	0.003065
dtype: float64	

Although the numbers are more accurate, the graphs are still better for non-data scientists to understand.

RandomForestRegressor

GradientBoostingRegressor



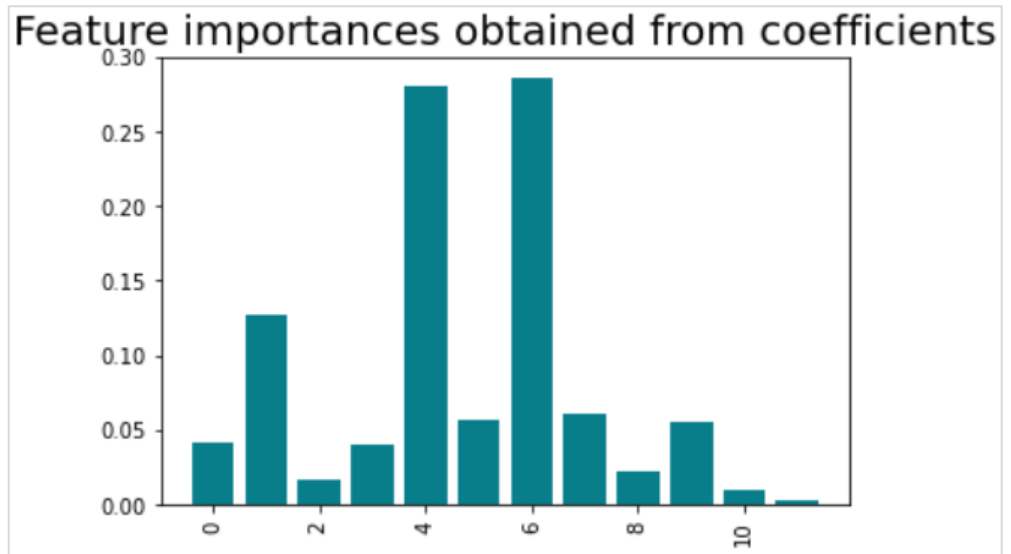
Although the difference is not obvious, there are still differences. According to the above graph, we know that the results of each model will be different. Perhaps because the theoretical differences between the two methods are not so great, the difference in the features importance is not great also. However, referring to the attachment, it can be found that in most cases, the feature importance of different models is very different, and some features are even considered as not contributing to training model.

RandomForestRegressor has the smallest MSE, but GradientBoostingRegressor has the smallest MAE. So I think the two models are very valuable.

Next, let's look at the best Classification model RandomForestClassifier and MLPClassifier.

RandomForestClassifier:

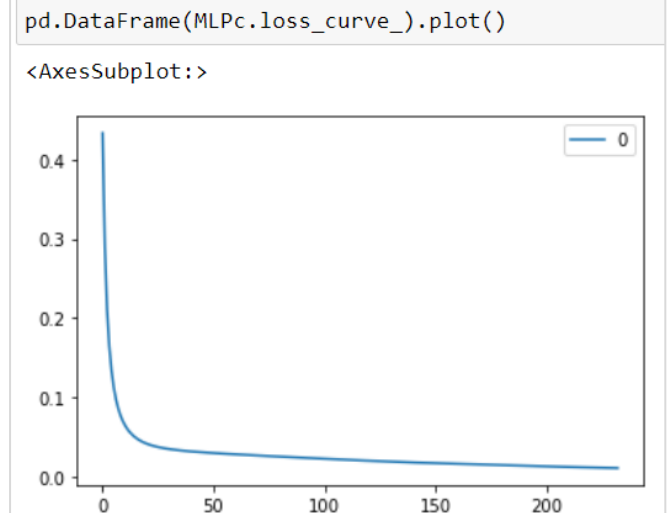
```
Feature importances
0      0.041752
1      0.127754
2      0.017240
3      0.039844
4      0.279727
5      0.057078
6      0.285862
7      0.060234
8      0.022079
9      0.055278
10     0.010235
11     0.002916
dtype: float64
```



For the Neural Network model, we can use the attribute “coefs_” of Classifier to help us understand the weight of each layer. In the neural network model, we can have many layers for analysis, and the weight of each feature to each layer is different. And we can know the weight matrix of each feature in each layer. we can use the attribute “loss_curve_” of Classifier to help us understand the changes of loss during training. Loss represents the difference between the predicted value and the actual value. The “loss_curve_” will have a different value with each epoch. So we can use this feature to visualize it to show the performance of our model.

MLPClassifier:

```
MLPc.coefs_
[array([[ 0.046, -0.032,  0.19 , ..., -0.118, -0.055, -0.585],
       [-0.2 ,  0.148, -0.217, ..., -0.055,  0.199, -0.192],
       [ 0.109,  0.539,  0.052, ...,  0.294,  0.2 ,  0.215],
       ...,
       [-0.17 ,  0.169, -0.203, ..., -0.01 , -0.089, -0.012],
       [-0.451, -0.145,  0.27 , ...,  0.092,  0.16 ,  0.202],
       [-0.274,  0.272,  0.027, ..., -0.261,  0.052,  0.176]]),
 array([[ -0.308],
       [ -0.308],
       [  0.188],
       [  0.449],
       [ -0.565],
       [ -0.296],
       [ -0.485],
       [ -0.23 ],
       [ -0.478],
       [  0.273],
       [  0.258],
       [  0.38 ],
       [  0.073]])]
```



The feature importance of the neural network model will vary with the layers, because the neural network model theory is like the neural network, which we cannot use simple diagram to show. So we have to use numbers to express feature importance, but the numbers are also difficult to explain the feature importance.

Another great attribute of neural networks is that you can look at the training level of its model and know if the model can be improved. Looking at the loss, we can see that the loss level is almost horizontal at the end, so we can know that the model has been well trained.

Application:

If we want to make predictions about future data, we also need to pre-process the data in the same way. Set the pre-processing of the data we decided to be a function to facilitate subsequent use. In this way, the code for data pre-processing is neat.

```
def dataset_preprocess(dataset):  
    #mapping Color  
    Color_mapping={"White":0,"Red":1}  
    dataset['Color[DV]'] = dataset['Color[DV]'].map(Color_mapping)  
  
    #preprocess  
    X = dataset[dataset.columns[:12]]  
    Y = dataset['Color[DV]']  
    from sklearn import preprocessing  
    z_score_scaler = preprocessing.StandardScaler()  
    X_z_score = z_score_scaler.fit_transform(X)  
    X=pd.DataFrame(X_z_score)  
    from sklearn.model_selection import KFold  
    kfold = KFold(n_splits=10,shuffle=True)  
    for train, test in kfold.split(dataset):  
        X_train =X.iloc[train]  
        Y_train =Y.iloc[train]  
        X_test =X.iloc[test]  
        Y_test =Y.iloc[test]
```

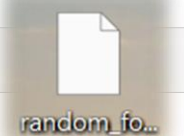
The reason I didn't set all the processing methods as functions at the beginning was because I didn't know what kind of data pre-processing would be the best. If all the pre-processing methods are saved as a function, the function will be quite messy. As long as we know which is the best after the comparison, we can save it as a function. In this way, we can have the cleanest function.

After importing the future dataset, assign the imported dataset as TheFutureDataset. Then use the function to pre-process the imported data set and assign it as a variable. Here we name this variable TFD.

```
TheFutureDataset=pd.read_csv('(File path and file name)')  
TFD=dataset_preprocess(TheFutureDataset)
```

Next, we need to save the model we just trained, then we can call it out for later use. The model storage tool introduced in scikit-learn is joblib. Therefore, we will use joblib. I save the model on the desktop, and after saving it, you can see the following icon on the desktop. Here we take the random forest classifier as an example.

```
import joblib  
joblib.dump(RFc, "C:\\Users\\88697\\Desktop\\random_forest.joblib")  
['C:\\Users\\88697\\Desktop\\random_forest.joblib']
```



In the future, we can import the model in the following ways:

```
loaded_rfc = joblib.load("C:\\Users\\88697\\Desktop\\random_forest.joblib")
```

Then, we can predict it in the following way:

```
probability=loaded_rfc.predict(TFd)
```

The above is how to save and apply the code. If we want to further deploy our results, we can use web pages or Application Programming Interface or even Application to deploy. LINE has a very simple API developer, so I can briefly introduce how we can achieve a beautiful deployment. This API is a chat bot, we can give data to the chat bot, and ask it to make predictions. We need to install the required applications such as git, Heroku CLI, etc. Save the code need to be save as a .py file. Use the deployment application Heroku on the terminal. Finally, use the integrated developer environment to deploy the code to Heroku.

Step1: Install the required programs


Install and register Heroku CLI, Git, Install gunicorn(pip install gunicorn).

Heroku is a free platform-as-a-service (PaaS). You can develop and deploy various websites on the Heroku platform on your own, allowing programs to run non-stop. At the same time, we also need Git to help us push the code into Heroku. Gunicorn, also known as the green unicorn (derived from the icon), is a Python web server gateway interface HTTP server. We need Gunicorn to make the code work properly on Heroku.

Create a folder that execute the heroku here.

In this way, we can ensure that the files are of the same level so that there will be no errors during operation.

Step 2: download the code into a .py file and save the file to the folder.



```
loaded_rfc = joblib.load("(Trained model file)")

def dataset_preprocess(dataset):
    #mapping Color
    Color_mapping={"White":0,"Red":1}
    dataset['Color[DV]'] = dataset['Color[DV]'].map(Color_mapping)
    #preprocess
    X = dataset[dataset.columns[:12]]
    Y = dataset['Color[DV]']
    from sklearn import preprocessing
    z_score_scaler = preprocessing.StandardScaler()
    X_z_score = z_score_scaler.fit_transform(X)
    X=pd.DataFrame(X_z_score)
    from sklearn.model_selection import KFold
    kfold = KFold(n_splits=10,shuffle=True)
    for train, test in kfold.split(dataset):
        X_train =X.iloc[train]
        Y_train =Y.iloc[train]
        X_test =X.iloc[test]
        Y_test =Y.iloc[test]

#To use LINEBot, a website server must be created. Here, the Flask module is used to create a website.
from flask import Flask
app = Flask(__name__)

from linebot import LineBotApi
from linebot.exceptions import InvalidSignatureError
from linebot.models import MessageEvent, TextMessage,TextSendMessage

#When the user sends a message to LINE Bot, the MessageEvent event will be triggered.
@handler.add(MessageEvent, message=TextMessage)
    message = TextMessage
    #means that the received text message, that is to say, the received text message will be processed by this routing.

#Then create a function to handle routing:
def handle_message(event):
    line_bot_api.reply_message(event.reply_token,TextSendMessage(text=event.message.text))
    #The parameter "event" contains the returned messages. For example, the name of the created function is handle_message

    if mtext == '(The data you want to predict)':
        TFD=dataset_preprocess(The data you want to predict)
        prediction=loaded_rfc.predict(TFd)
        #The syntax of the return message is:
        line_bot_api.reply_message(event.reply_token,prediction)

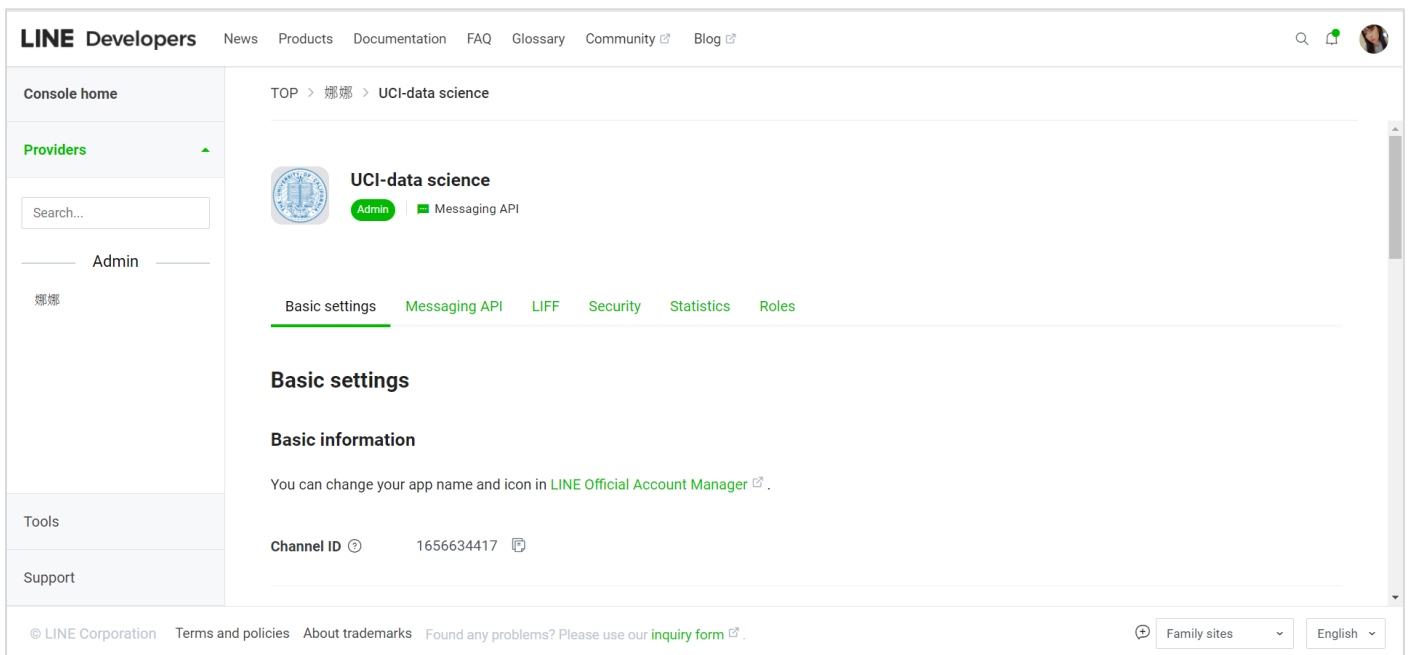
if __name__ == '__main__':
    app.run()
```

Because we have to run many different files at the same time, and some files cannot work on jupyter notebook, so we must organize the code and download the code, which can be used it on the IDE. Let the IDE integrated these different files and run at the same time.

I didn't finish the code properly, because I just want to show the method.

Step 3: Go to the API development interface of LINE to create an API

We must build an API to allow the code to have an interface that can be displayed.



And set LINE Bot setting: Profile (tell Heroku how to start the app), requirements.txt (Tell Heroku what other packages are needed for the environment of this app), runtime.txt (tell Heroku the python version specified by this app).

Step 4: Use the terminal to execute Heroku

```
heroku login
```

```
heroku create (appname)
```

We use terminals to maintain Heroku's continuous operation. We can also use other executor to operate, but I think the terminal is the simplest and fastest.

Step 5: Use the IDE to deploy the code to Heroku

1.Create a new Git repository (in terminal)

```
git init
```

```
heroku git:remote -a (appname)
```

2.Deploy your application

```
git add.
```

```
git commit -am "make it better"
```

```
git push heroku master
```

We use IDE to help us gather all the elements together and work together.

Step 6: Let LINE run the code

Paste the website deployed by Heroku in the Webhook URL field in LINE Developers.(Add "/callback" to the URL).

Webhook URL ⓘ	<input type="text" value="Enter your app's webhook URL"/>
---------------	---

Summary and Conclusion

We have discussed our business purpose earlier. We want to know the quality of wine accurately and make commercial distinctions, and we want to know the special types of wine accurately to develop new products.

Then we can use some code and some charts to help us understand this data set. When we understand what kind of distribution the data is and learn whether the data is clean or not, then we will know how we need to deal with this data set. Such as, clean up data, deal with null values, decide whether the data needs to be normalized or standardized, Dealing with data imbalances, etc.

The dataset is very clean, so there is not much data pre-processing we can do. Generally speaking, the actual dataset should be very dirty, so there can be many different processing methods, so there will be many different model performances. But this data is very clean, so we can't show how we pre-process the data creatively. But I still try my best to try different data pre-processing methods.

In fact, we won't know what kind of data pre-processing method is the best before we compare all the model results together. Even if the data pre-processing method we think of will be nice for the dataset, it does not mean that the model results run out later will be the best. Therefore, each of data pre-processing methods must trained the model individually.

When training the model, the most time-consuming part is that all the 128 models have to adjust the parameters one by one, because I still don't know what the parameters of each model mean, so I use try and error methods to find the best parameters. In addition, my computer performance is not good enough, so after the pre-processing of SMOTE, some models cannot complete the training.

About model evaluation methods, there are many methods. I use the most common ones. There are more direct way to know the performance of the data. S Since different data pre-processing methods are distinguished by individual files, there is no likely to use code to help me organize all the model results. So I made a form in Word which all our model results can be recorded in it. So we can make comparisons more efficiently.

As for the deployment part, I can simply visualize the results of our data science, and then save the model so that subsequent programs can use it. We visualize the model we ran to present our results. If it is not a neural network model, we can get the importance of features, which means that we can know the importance of each feature to the model. In the neural network model, we can have many layers of analysis results, and the weight of each layer is different. And we can know the weight matrix of each feature in each layer. In addition, we can also use the loss to observe the process of his training. Or we can export the trained model for subsequent use by other departments. We can even also do it more progressively and deploy with a beautiful interface. I used the API of a chatbot as a demonstration in the previous section.

After we show our results, other departments can put forward plans for the company based on the results we have shown to increase the company's revenue. For example, can we differentiate products so that the company's sales policy can be changed to increase revenue, or can we develop new products which let the company have a new source of income?