GOAL: To predict the quality of Wine based on certain parameters.

Data Source: Red Wine Quality Dataset - Kaggle (https://www.kaggle.com/datasets/uciml/red-wine-quality-cortez-et-al-2009)

We start by importing the data. { Required Library: Pandas }

In [1]:
```
import pandas as pd
data = pd.read_csv(r"C:\Users\goura\Desktop\Data Science\Datasets\red-wine.csv")
data.head()
```

Out[1]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphate
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.6
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.6
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.5
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.5

Lets get some info on the table.

In [2]:
```
data.columns
```

Out[2]:
```
Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
       'pH', 'sulphates', 'alcohol', 'quality'],
      dtype='object')
```

In [3]:
```
data.describe()
```

Out[3]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	
count	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1
mean	8.319637	0.527821	0.270976	2.538806	0.087467	15.874922	
std	1.741096	0.179060	0.194801	1.409928	0.047065	10.460157	
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	
25%	7.100000	0.390000	0.090000	1.900000	0.070000	7.000000	
50%	7.900000	0.520000	0.260000	2.200000	0.079000	14.000000	
75%	9.200000	0.640000	0.420000	2.600000	0.090000	21.000000	
max	15.900000	1.580000	1.000000	15.500000	0.611000	72.000000	

In [4]:
```
data['quality'].value_counts()
```

Out[4]:
```
quality
5    681
6    638
7    199
4     53
8     18
3     10
Name: count, dtype: int64
```

We now know about important statistics of all our attributes, as well as the label we want to predict - "quality"

Visualizing the data is important to gain insights about our data

For this, we can use: seaborn, matplotlib, pandas.plotting

An important information about each attribute is the correlation of its to the Label, as well as other attributes.

In [5]:
```
corr_matrix = data.corr()
corr_matrix
```

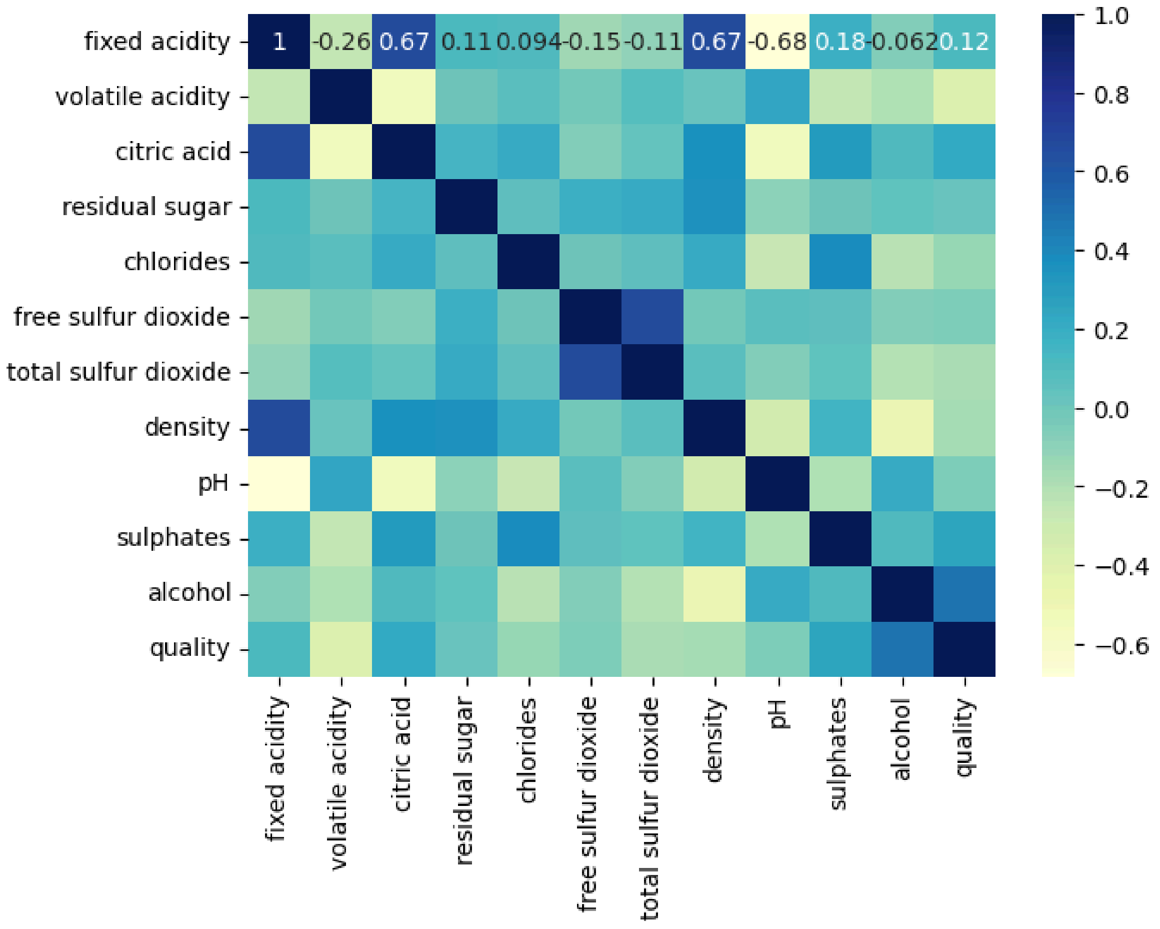
Out[5]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	
fixed acidity	1.000000	-0.256131	0.671703	0.114777	0.093705	-0.153794	-0.113181	(
volatile acidity	-0.256131	1.000000	-0.552496	0.001918	0.061298	-0.010504	0.076470	(
citric acid	0.671703	-0.552496	1.000000	0.143577	0.203823	-0.060978	0.035533	(
residual sugar	0.114777	0.001918	0.143577	1.000000	0.055610	0.187049	0.203028	(
chlorides	0.093705	0.061298	0.203823	0.055610	1.000000	0.005562	0.047400	(
free sulfur dioxide	-0.153794	-0.010504	-0.060978	0.187049	0.005562	1.000000	0.667666	-(
total sulfur dioxide	-0.113181	0.076470	0.035533	0.203028	0.047400	0.667666	1.000000	(
density	0.668047	0.022026	0.364947	0.355283	0.200632	-0.021946	0.071269	1
pH	-0.682978	0.234937	-0.541904	-0.085652	-0.265026	0.070377	-0.066495	-(
sulphates	0.183006	-0.260987	0.312770	0.005527	0.371260	0.051658	0.042947	(
alcohol	-0.061668	-0.202288	0.109903	0.042075	-0.221141	-0.069408	-0.205654	-(
quality	0.124052	-0.390558	0.226373	0.013732	-0.128907	-0.050656	-0.185100	-(

We can also plot this using a heatmap

In [6]:
```python
import seaborn as sns
import matplotlib.pyplot as plt
plt.figure(figsize=(7,5))
sns.heatmap(corr_matrix,annot=True,cmap='YlGnBu')
```

Out[6]:
```
<Axes: >
```



The diagonal is to be ignored, as it signifies the correlation of one attribute to itself, which is, without exception, always 1 { highest possible value }. In this plot, the darker colours signify a higher correlation.

Now that we have visualized the data and have improved our understanding about it, we can proceed to pre-process the data for fitting into the desired model.

In [7]:
```
features = data.columns[:-1].values          #all attributes except quality
label = data.columns[-1]                     #quality
```

It is good practice to split the available data into two parts, Train Set & Test Set. This is to evaluate the model trained on the Train Set by testing it on the Test Set. Otherwise, a model tested on the data its been trained on will give exceptional results, but may fail provide acceptable results when unknown data is introduced.

This tast has been made easier by Scikit-Learn, due to availability of: Random Sampling, Stratified Sampling

Random Sampling picks out Data Randomly, while Stratified Sampling picks out data which represents the Dataset as a whole. So, Stratified Sampling provides us with better data to train.

In [8]:
```
from sklearn.model_selection import StratifiedShuffleSplit
split = StratifiedShuffleSplit(n_splits=1,test_size=0.2,random_state=42)
for train_index,test_index in split.split(data,data['quality']):
    train_set = data.loc[train_index]
    test_set = data.loc[test_index]
```

In [9]:
```
train_set
```

Out[9]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulph
1542	6.7	0.855	0.02	1.90	0.064	29.0	38.0	0.99472	3.30	
1558	6.9	0.630	0.33	6.70	0.235	66.0	115.0	0.99787	3.22	
344	11.9	0.570	0.50	2.60	0.082	6.0	32.0	1.00060	3.12	
924	8.6	0.470	0.27	2.30	0.055	14.0	28.0	0.99516	3.18	
971	10.4	0.260	0.48	1.90	0.066	6.0	10.0	0.99724	3.33	
...	...	...	...	...	...	...	...	...	...	
1056	8.9	0.480	0.53	4.00	0.101	3.0	10.0	0.99586	3.21	
1394	6.4	0.570	0.14	3.90	0.070	27.0	73.0	0.99669	3.32	
337	7.8	0.430	0.32	2.80	0.080	29.0	58.0	0.99740	3.31	
539	11.2	0.500	0.74	5.15	0.100	5.0	17.0	0.99960	3.22	
1083	8.7	0.420	0.45	2.40	0.072	32.0	59.0	0.99617	3.33	

1279 rows × 12 columns

In [10]:
```
test_set
```

Out[10]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulph
963	8.8	0.27	0.39	2.0	0.100	20.0	27.0	0.99546	3.15	
475	9.6	0.68	0.24	2.2	0.087	5.0	28.0	0.99880	3.14	
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	
329	10.7	0.46	0.39	2.0	0.061	7.0	15.0	0.99810	3.18	
149	8.2	0.40	0.44	2.8	0.089	11.0	43.0	0.99750	3.53	
...	...	...	...	...	...	...	...	...	...	
1173	7.6	0.36	0.31	1.7	0.079	26.0	65.0	0.99716	3.46	
827	7.1	0.46	0.14	2.8	0.076	15.0	37.0	0.99624	3.36	
356	11.5	0.41	0.52	3.0	0.080	29.0	55.0	1.00010	3.26	
1287	8.0	0.60	0.08	2.6	0.056	3.0	7.0	0.99286	3.22	
789	8.6	0.63	0.17	2.9	0.099	21.0	119.0	0.99800	3.09	

320 rows × 12 columns

In [11]:
```
wine_features_tr = train_set[features]
wine_label_tr = train_set["quality"]
wine_features_te=test_set[features]
wine_label_te=test_set["quality"]
```

Next, we need to handle:

-> Null values [imputing / deletion]

-> Categorical values [encoding]

Since our dataset doesnt have any categorical values, we dont have to worry about encoding.

For null values, all rows with null values can be dropped by using the dropna() function, but data is lost in that process. A better approach is the to fill in the null values using any parameter from the other values.

This can be done using the SimpleImputer from sklearn library.

Lets first check no of null values

In [12]:
```
wine_features_tr.isna().sum()
```

Out[12]:
```
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
dtype: int64
```

We have no null values, so we dont need Imputation either.

As we see, all the attributes here are numeric values, so we can use either of these three:

-> LinearRegressor

-> DesicionTreeRegressor

-> RandomForestRegressor

Lets start with LinearRegressor

In [13]:
```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(wine_features_tr,wine_label_tr)
lin_pred = lin_reg.predict(wine_features_te)
lin_pred
```

```
Out[13]: array([6.32922267, 5.46660305, 5.1572968 , 5.5688065 , 5.58138071,
       6.03262132, 5.08478005, 4.4996503 , 5.51573986, 6.12624362,
       5.13294042, 5.47274026, 5.23882231, 5.23447299, 5.4457214 ,
       5.80529027, 5.24443432, 5.00866933, 6.5864237 , 6.08641721,
       6.03212294, 5.78202105, 5.83143791, 5.35938226, 5.05522209,
       5.52541505, 5.34617197, 5.48826587, 6.60415795, 5.22823928,
       5.87448099, 5.337187  , 6.29414645, 5.79910105, 5.97181256,
       5.32392074, 5.32076459, 6.12007723, 4.91697379, 5.9780407 ,
       4.99184165, 5.11798898, 5.50741004, 6.47789429, 6.34880026,
       5.52113666, 6.35516118, 5.9295155 , 5.44363406, 5.39063968,
       5.06639384, 6.40676754, 5.15723593, 5.93563782, 6.0923704 ,
       5.00420671, 5.01670795, 5.14684463, 5.46842186, 5.46475082,
       5.4149858 , 5.99980307, 5.38612728, 5.93670629, 6.21774091,
       5.42973318, 5.93922809, 5.42879083, 5.03596587, 5.69424338,
       6.22102553, 5.17240727, 5.67461424, 5.80940751, 5.20339628,
       6.04314157, 5.6944099 , 5.94002034, 5.02275907, 6.00783492,
       5.50288513, 5.61776048, 5.70449731, 5.18933431, 4.89542771,
       5.13927392, 5.37291913, 5.29283133, 6.741597  , 5.49011852,
       5.31546021, 5.27965006, 5.64205928, 6.9233452 , 5.42661408,
       5.86596115, 4.92318478, 5.22794257, 6.16990227, 4.92510602,
       6.45727047, 6.19164181, 6.19880102, 6.19988056, 5.8089154 ,
       6.36287705, 5.91774623, 6.08340607, 6.51102548, 6.03517165,
       5.13258547, 5.2487853 , 5.25044838, 5.33481883, 5.78505715,
       5.75572629, 5.2273458 , 5.21305364, 6.35516118, 6.10715405,
       6.38478885, 5.19731595, 6.12693509, 5.96089635, 5.70645191,
       5.9316095 , 5.92777718, 5.0500635 , 5.14323594, 6.2314931 ,
       6.00276765, 5.5370209 , 4.73241012, 5.67879703, 6.17866859,
       6.29414645, 5.1417121 , 6.04066333, 4.59771064, 6.2583191 ,
       5.03393208, 5.3343219 , 5.34905505, 5.15614103, 4.84024259,
       6.24755999, 5.2313332 , 6.03403867, 5.53029969, 5.69786056,
       5.89057799, 5.05686756, 5.06886197, 4.82395697, 5.45823316,
       5.18186828, 5.12723153, 4.96567289, 6.2216525 , 6.29723548,
       5.8535305 , 5.00480526, 5.56120841, 6.29840662, 5.17061   ,
       6.35533587, 5.69960638, 6.00630862, 5.14187862, 5.78922037,
       5.97313636, 5.31518246, 5.30927815, 6.4075138 , 6.9525867 ,
       5.26805146, 5.75367651, 5.28589675, 5.67837007, 6.60001512,
       5.57741169, 4.96312471, 5.99089229, 5.37286306, 5.37173079,
       5.75779603, 6.02555038, 4.9760125 , 5.46423039, 5.67178098,
       6.09379253, 5.10858798, 5.32680674, 5.11562722, 5.92901382,
       6.38054345, 6.37548313, 5.22935395, 4.88345135, 5.50073284,
       6.31665279, 5.30152965, 5.29223583, 4.97985461, 5.7713188 ,
       5.4972377 , 5.362902  , 5.70273472, 6.49624636, 5.98092583,
       5.36800443, 5.78164128, 5.02275907, 5.65822505, 6.42701669,
       5.43256692, 6.11988775, 5.83018315, 5.77600717, 6.06139467,
       6.02058756, 5.67232135, 5.44920694, 4.98855493, 4.92960265,
       6.17629877, 5.75572629, 6.02344507, 6.5864237 , 5.60562418,
       5.77369916, 5.40043274, 5.79457703, 6.18285543, 5.30277283,
       5.89910612, 5.2742837 , 6.14243028, 5.74266313, 5.52365247,
       5.55407538, 5.12988101, 5.38659488, 6.04768967, 5.1186386 ,
       5.58344447, 6.47744101, 6.26384766, 5.72134582, 6.11797126,
       5.42668873, 5.33973841, 4.82683796, 5.2668879 , 5.72198966,
       5.46660305, 5.20907819, 5.39514563, 5.05065068, 5.67943446,
       5.44028485, 5.72756533, 6.18493254, 4.933566  , 6.51606728,
       5.96587438, 5.18773018, 5.8564102 , 4.63205822, 6.11767705,
       5.14053459, 5.44660467, 4.80313324, 6.02147879, 5.07904335,
       6.71855541, 5.73230137, 4.83393384, 5.93922809, 5.29581816,
       5.84255815, 5.3534057 , 5.29406337, 5.66982931, 6.16319229,
       5.82810736, 5.83398351, 5.34194927, 5.43673221, 5.43067462,
       5.90403184, 6.38478885, 5.67135789, 5.46364921, 5.04065991,
       5.39389289, 5.73661926, 5.21232179, 5.20972972, 6.60203419,
       5.2303544 , 5.72699582, 5.16708345, 5.60304828, 5.85757349,
       5.57249085, 5.50854071, 5.38430579, 5.86251156, 6.40980961,
       5.43804816, 5.39434265, 5.27138491, 5.45811079, 5.31057753,
       5.41993909, 5.70312574, 6.15126939, 6.31773972, 4.96557987])
```

Just looking at the prediction doesnt give us much info about the performance of the model. We can check the performance by using:

-> Mean Absolute Error

-> Mean Squared Error

We can apply these to compare predicted values to actual values between train and test set, or we can cross validate.

```
In [14]: from sklearn.metrics import mean_squared_error
         from sklearn.metrics import mean_absolute_error
         mean_squared_error(wine_label_te,lin_pred)
```

```
Out[14]: 0.40636065372564334
```

```
In [15]: mean_squared_error(wine_label_tr,lin_reg.predict(wine_features_tr))
```

```
Out[15]: 0.4206571060060278
```

Not much deviation between MSE of training and testing data signifies that there hasn't been overfitting.

Lets Try out a different model : DesicionTreeRegressor

```
In [16]: from sklearn.tree import DecisionTreeRegressor
         tree_reg = DecisionTreeRegressor()
         tree_reg.fit(wine_features_tr,wine_label_tr)
         tree_pred = tree_reg.predict(wine_features_te)
         mean_squared_error(wine_label_te,tree_pred)
```

```
Out[16]: 0.5875
```

Thats high! What could've been the reason?

```
In [17]: mean_squared_error(wine_label_tr,tree_reg.predict(wine_features_tr))
```

```
Out[17]: 0.0
```

As we can see, the MSE for training data is 0. Which means, the model has been overfitted with data.

There are two possible measures in this scenario:

-> Try another model

-> Change the parameters { currently set to default }

We will try the first option first. Lets use an ensemble model { model built upon another model } called he RandomForestRegressor

In [18]:
```
from sklearn.ensemble import RandomForestRegressor
forest_reg = RandomForestRegressor()
forest_reg.fit(wine_features_tr,wine_label_tr)
forest_pred = forest_reg.predict(wine_features_te)
mean_squared_error(wine_label_te,forest_pred)
```

Out[18]:
```
0.3229634375
```

Thats a huge improvement in MSE {-0.8}! The previous best was 0.41.

Lets check for overfitting, even if its unlikely given the better prediction outcome.

In [19]:
```
mean_squared_error(wine_label_tr,forest_reg.predict(wine_features_tr))
```

Out[19]:
```
0.0481591086786552
```

A very low error! Does this mean slight overfitting is present?

We can try out changing parameters, but before that, Is there a better method to compute the errors?

We can use cross validation for determining the performance of each of the model over the entire dataset.

In [20]:
```
from sklearn.model_selection import cross_val_score
def display_scores(scores):
    print("Scores: ",scores)
    print("Mean: ",scores.mean())
    print("Std. Deviation: ",scores.std())
#score = cross_val_score(model_name,feature_names,label_name,scoring_criteria,cv
```

In [21]:
```
display_scores(-cross_val_score(lin_reg,wine_features_tr,wine_label_tr,scoring =
```

```
Scores:  [0.56364537 0.4429824  0.38302744 0.40166681 0.29687635 0.37322622
 0.33184855 0.50182048 0.51661311 0.50468542]
Mean:  0.4316392172121962
Std. Deviation:  0.08356359730413934
```

In [22]:
```
display_scores(-cross_val_score(tree_reg,wine_features_tr,wine_label_tr,scoring
```

```
Scores:  [0.640625   0.71875    0.5625     0.546875   0.453125   0.5546875
 0.609375   0.765625   0.8046875  1.03149606]
Mean:  0.6687746062992126
Std. Deviation:  0.15871731115385232
```

In [23]:
```
display_scores(-cross_val_score(forest_reg,wine_features_tr,wine_label_tr,scorin
```

```
Scores:  [0.37359453 0.41460156 0.28772344 0.32830859 0.21892344 0.29854922
 0.27300313 0.39192187 0.43314844 0.46308583]
Mean:  0.34828600455216535
Std. Deviation:  0.07501056638032301
```

By a significant margin, The RandomForestRegressor is performing the best. Few parameters of RandomForestRegressor are

-> n_estimators

-> max_depth

-> min_samples_split

Now, to imrove the performance, we should try changing some parameters to check for better results. But this task is difficult to do by man.

So, Scikit-Learn has a feature "GridSearchCV" which helps us with this

In [24]:
```
from sklearn.model_selection import GridSearchCV
param_grid = [
    {'n_estimators':[100,150,200],'max_depth':[None,15],'min_samples_split':[2,1
]
grid_search = GridSearchCV(forest_reg, param_grid, cv = 5, scoring = "neg_mean_s
grid_search.fit(wine_features_tr,wine_label_tr)
grid_search.best_params_
```

Out[24]:
```
{'max_depth': None, 'min_samples_split': 2, 'n_estimators': 150}
```

This tries out a total of 36 models so it consumes significant amount of time

As it says, the best possible combination has been obtained. Lets fit it into our model.

In [25]:
```
forest_reg2 = RandomForestRegressor(max_depth=None, min_samples_split=2, n_estim
display_scores(-cross_val_score(forest_reg2,wine_features_tr,wine_label_tr,scori
```

```
Scores:  [0.38590937 0.42523301 0.28973281 0.32364336 0.22171309 0.29713711
 0.26907695 0.38872656 0.44466172 0.45990768]
Mean:  0.35057416615403554
Std. Deviation:  0.07720999973058873
```

As we can see, even with parameters, the error is approximately the same. Thus we go ahead with the no parameter model to save computational time.

Now then, what is the accuracy of our model? Lets check.

In [26]:
```
errate = (-cross_val_score(forest_reg,wine_features_tr,wine_label_tr,scoring = "
1-errate
```

Out[26]:
```
0.9386207770403946
```

Thus, the accuracy of the model would be,

1-errate ~ 0.94 = 94%

Now that we have created and tested the model, lets create the input function

```
In [28]: lst=[i for i in train_set.columns[:-1].values]
         while True:
             print("Do you want to make predictions? (Y/N)")
             a = input()
             if a=="N":
                 break
             elif a!="Y" and a!="N":
                 print("Wrong input")
             elif a=="Y":
                 val = []
                 for i in range(11):
                     val.append(float(input(f"Enter value of {lst[i]}: ")))
                 df = pd.DataFrame([val], columns=lst)
                 pred = forest_reg.predict(df)
                 print(pred)
```

```
Do you want to make predictions? (Y/N)
[5.3]
Do you want to make predictions? (Y/N)
[5.89]
Do you want to make predictions? (Y/N)
```