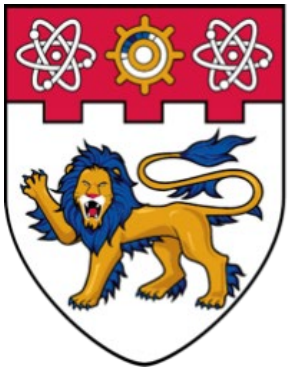
`



NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CB4247 Statistics & Computational Inference to Big Data for Chemical and Biomedical Engineers

Professor Yin Xunyuan

Individual Project (30%)

Name: Lim Junwei Darien

Matriculation No. : U1921811F

Table of Contents

Executive Summary

Red wine quality data set of the Portuguese "Vinho Verde" wine with 11 physicochemical variables and its corresponding quality was used to train and test on 2 regression models, linear least square and gaussian process regression. Gaussian process regression model with these kernels, whitekernel & dotproduct show better sum of squared error and mean squared error metrics than linear least square despite having 1% reduction in R^2 value. In conclusion, gaussian process regression model is a better model than linear least square in explaining the nature of the data set and to provide better prediction of red wine quality, that would enable the reduction of human sensory test to provide the red wine quality, thus reducing cost and increase efficiency.

1. Introduction

1.1 Dataset

Red wine quality data set of the Portuguese "Vinho Verde" wine was obtained from UCI machine learning repository. 11 physicochemical variables were recorded for 4898 times which are fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, alcohol. The quality of the wine was judged based on human sensing judgment which has a range from 1 to 10 that corresponds to the 11 physicochemical variables. The dataset obtained was used in 2 regression models, linear least-squares, and Gaussian process regression.

1.2 Regression Model

1.2.1 Linear least-squares

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_k x_k + e$$

where θ is the estimator required to be estimated; $x_1 \ldots x_x$ are the regressors; e is the equation error

This model works by minimizing the sum of error squares that would give the estimated θ. It is under the assumption of "e" is normal distributed, zero mean and has constant variance across all observations. This would mean the least square estimator, $\hat{\theta}$ is unbiased and is close or equal to θ . The

equation above could be manipulated to give us the equation to find sum of error squares that is shown below:

$$\min_{\theta_1 \dots \theta_k} \sum e^2 = \sum (y - \hat{\theta}_0 - \hat{\theta}_1 x_1 \dots \theta_k x_k)^2 = (Y - X\hat{\theta})^T (Y - X\hat{\theta})$$

Where Y, X are expressed in matrix form

$\hat{\theta}$ can be estimated using:

$$\hat{\theta} = (X^T X)^{-1} X^T Y$$

The table below shows the advantages and disadvantages of using least square regression:

Advantages	Disadvantages
Simple and model is easily understood	Sensitive to outliers
model training and prediction is fast	Assumes a linear relationship between dependent and independent variable which are usually not true

Table 1.1 Advantages and disadvantages of linear least squares

1.2.2 Gaussian process regression

Gaussian process (GP) describes probability distribution using functions, since there is no information about the input data thus for prior belief of the function, 0 is usually assumed for the mean. The possible functions that fit the characteristics of input data for eg. linear, periodic, noisy and etc is also known as kernels in Gaussian process regression [1].

The table below shows the advantages and disadvantages of using Gaussian process regression [1,2]:

Advantages	Disadvantages
Maintain high certainty of prediction	Computationally Expensive as GP are non-parametric, all the training data is considered every time a prediction is made
Shaping of prior belief through different kernels	Loses efficiency in high dimensional spaces, usually when the number of features is over a few dozens

Table 1.2 Advantages and disadvantages of Gaussian process regression

Kernels are also called covariance functions, they provide the assumptions on the function by defining the covariances of 2 data points. Stationary kernels depend only on the Euclidean distance of 2 data points, "$d(x_i,x_j)$" and not on their absolute values. For our context, x_j is the output data, quality of wine and x_i represents all other inputs. [1]

$$k(x_i, x_j) = k(d(x_i, x_j))$$

Types of Kernels available:

1) Constant kernel: helps to scale the magnitude of other factor(kernel), where it modifies the mean of the gaussian process through the parameter *constant_value*. [2] It's defined as:

$$k(x_i, x_j) = constant\_value \; \forall \; x_1, x_2$$

2) WhiteKernel: part of a sum kernel where it explains the noise component of the signal. [2]

3) Radial Basis function (RBF) Kernel: Is a stationary kernel when used can smooth out the regression. It is parameterized by a length-scale parameter $\iota>0$ and is defined as:

$$k(x_i, x_j) = \exp\left(-\frac{d(x_i, x_j)^2}{2l^2}\right)$$

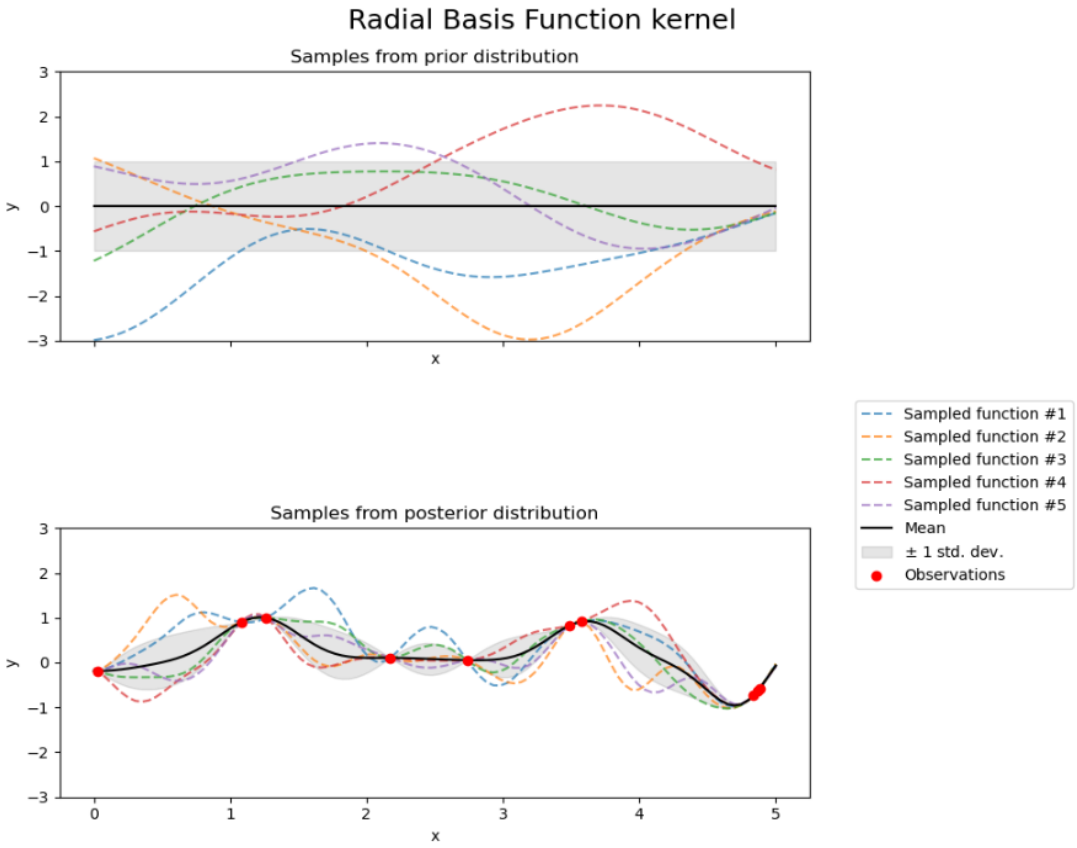


Figure 1.1 Results of GP regression from RBF kernel [2]

4) Rational Quadratic: It is an infinite sum of RBF kernels with different characteristics length scales, it is also parameterized by length-scale parameter $l>0$ and a scale mixture parameter $\alpha>0$. It is defined as:

$$k(x_i, x_j) = \left(1 + \frac{d(x_i, x_j)^2}{2\alpha l^2}\right)^{-\alpha}$$

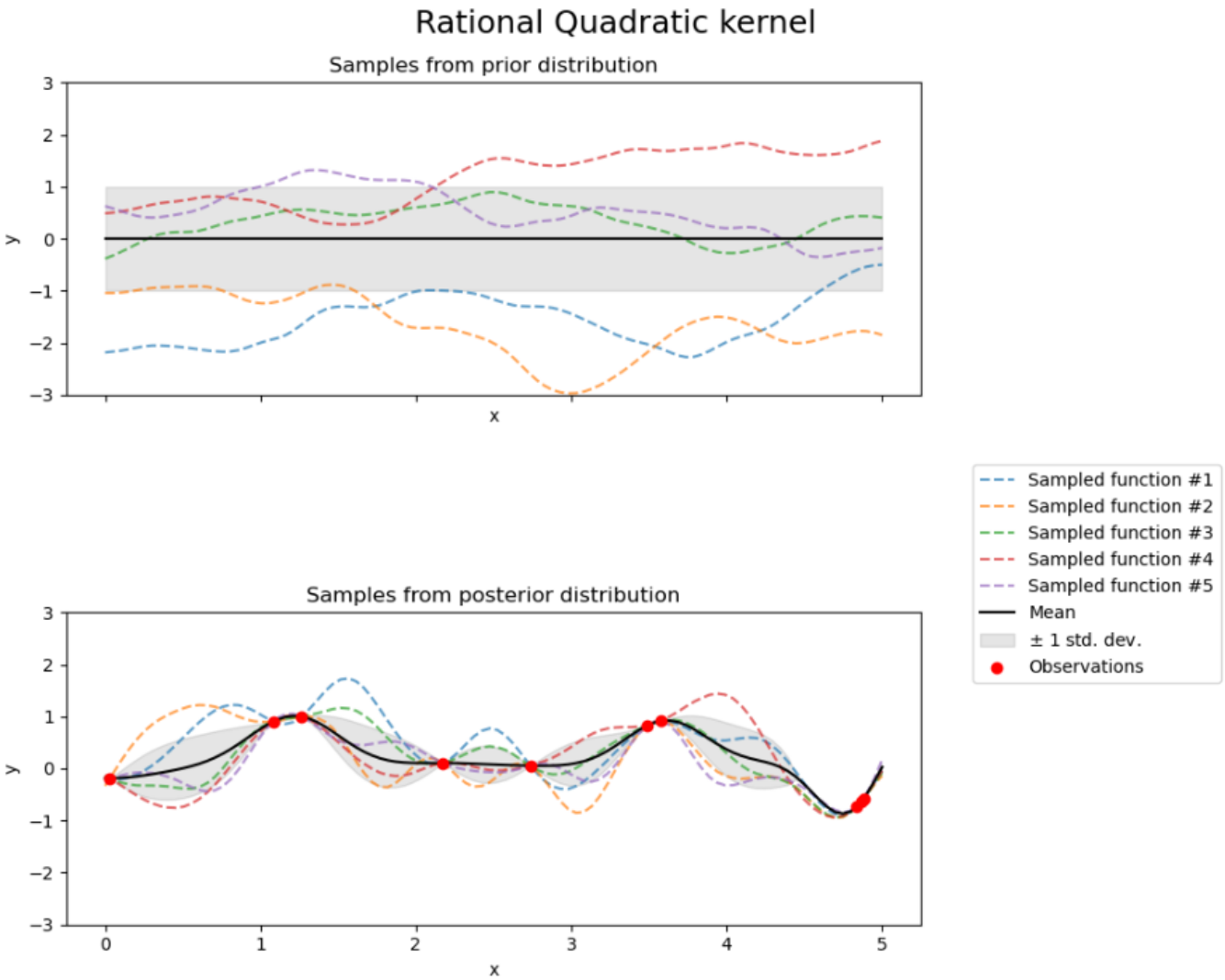


Figure 1.2 Results of GP regression from Rational Quadratic kernel [2]

5) Dot-Product: when x_i and x_j covariance is calculated through $x_i \cdot x_j$, this is called dot product covariance function. It is parameterized by σ_0^2, and defined as:

$$k(x_i, x_j) = \sigma_0^2 + x_i \cdot x_j$$

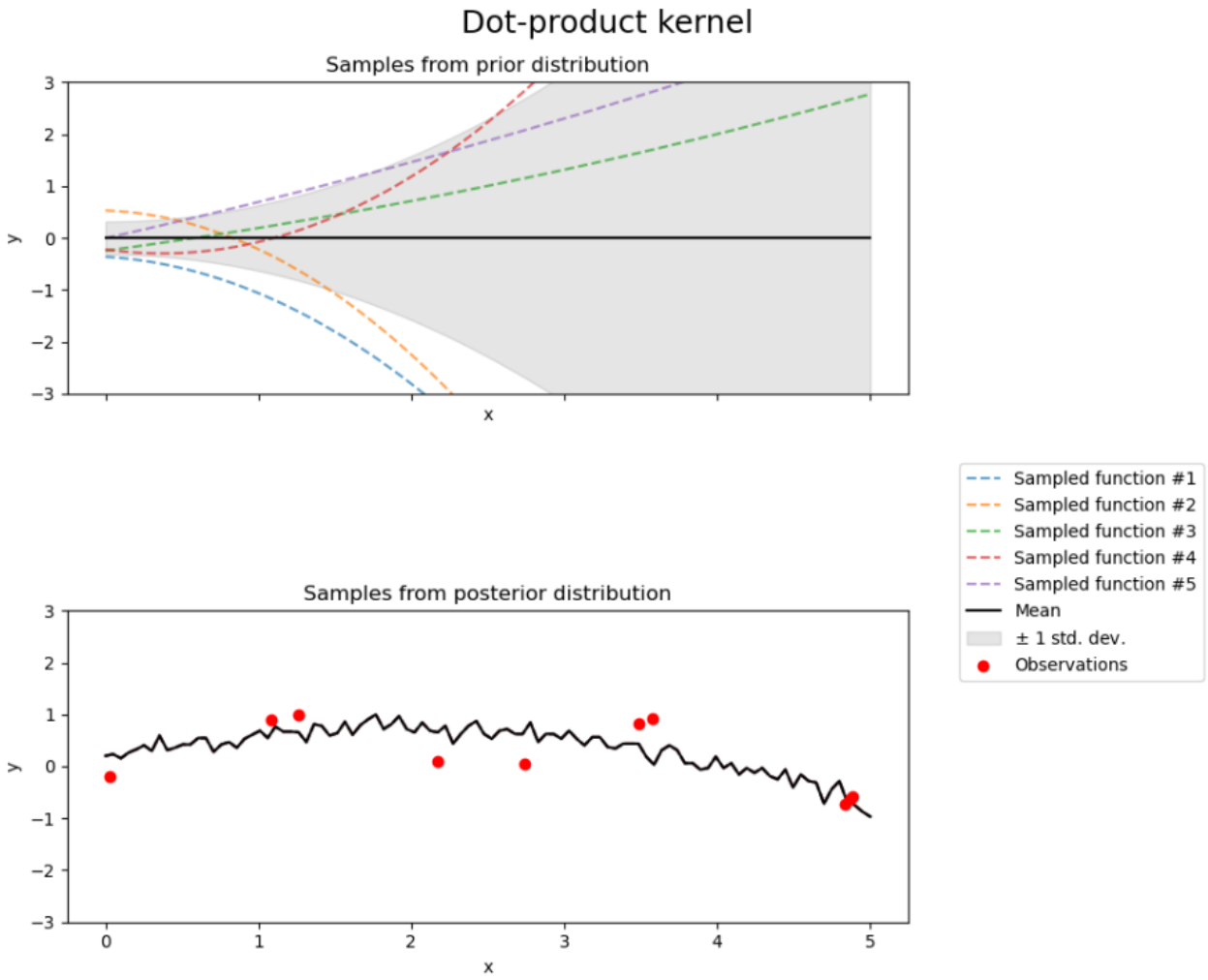


Figure 1.3 Results of GP regression from Dot Product kernel with exponent 2. [2]

Kernels can be combined to further described the relationship between the input and output data when there is more than one type of features. Multiplying kernels acts as an AND operation, if 2 kernels are multiplied together the resulting kernel will have high covariance if both kernels have high covariance. Adding of kernels acts as an OR operation, resulting kernel will have high covariance if either one of the 2 kernels have a high covariance.[3]

Illustrated as below:

Addition of $k_{sum}(X,Y) = k_1(X,Y) + k_2(X,Y).$ kernels:

Multiplying of kernels: $k_{product}(X,Y) = k_1(X,Y) * k_2(X,Y)$

2. Modelling & Data Processing

Dataset was checked for null values, outliers for its 11 physicochemical variables.

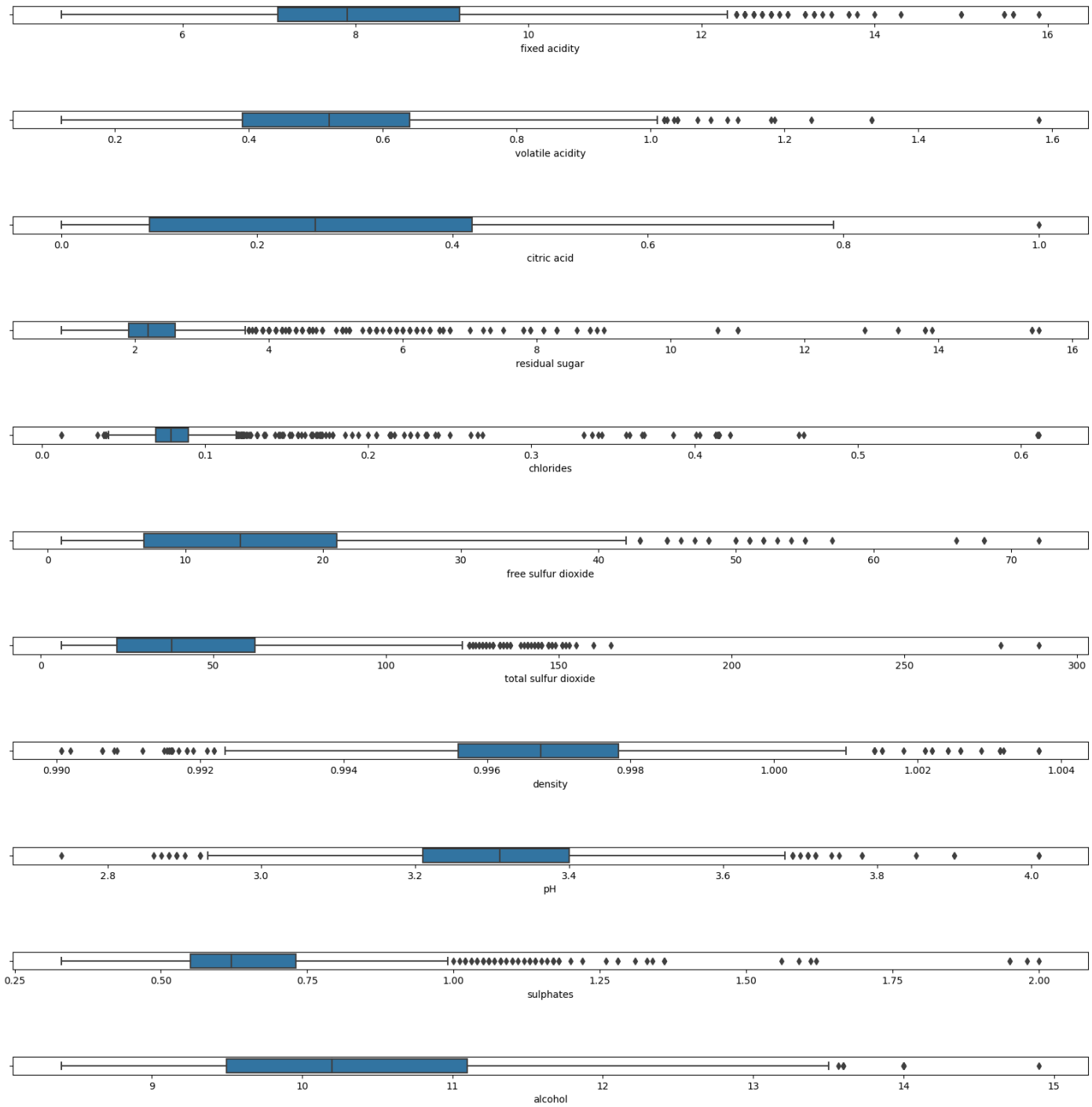


Figure 2.1 Box plot of the variables

From Figure 2.1, it shows that most of the variables have outliers that could be considered to remove to improve the regression model, however in this context as the data points are significant to the studies and removing these extreme values force the subject area to look less variable than it is in reality. Thus, no outliers were removed, and no data recording error was assumed.

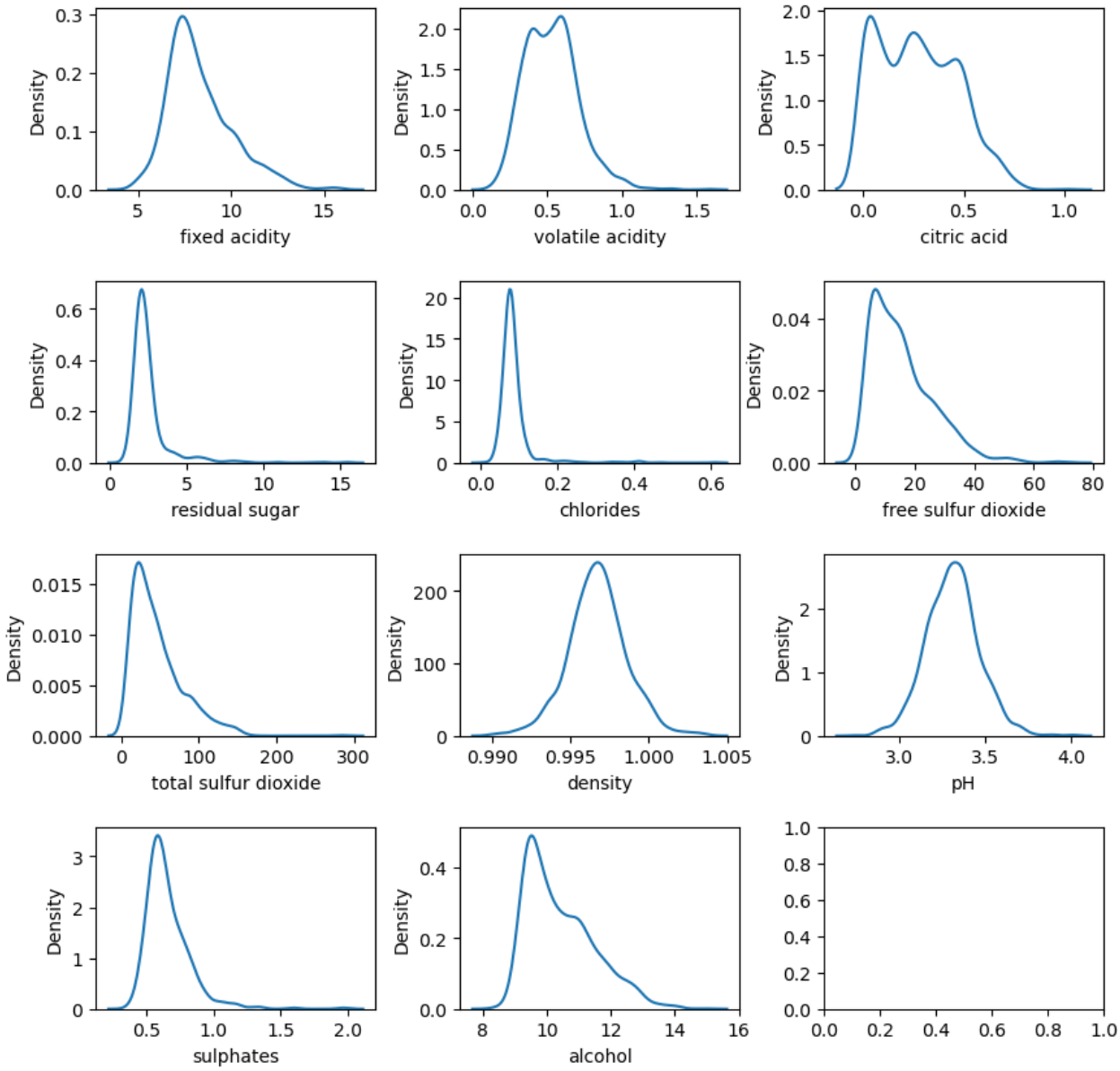


Figure 2.2 Density plot of the input data

From figure 2.2, some of the variables have skewed data which corresponds to the boxplot in figure 4, and this would lower the effectiveness of some statistical model. Data transformation would be required to normalize the data since removing the outlier is undesirable in this context.

2.1 Linear Least Squared

Linear least squared model is trained with 2 different datasets, untransformed and transformed by box cox to tackle the vast number of outliers so that the data is more normally distributed.

Untransformed and transformed Dataset was split into 60% and 40% of training and test datasets.

Sklearn library was used from python and the training set was input into the “LinearRegression” function as shown below:

```
X_tr = X_train.iloc[:, :i].copy()
X_te = X_test.iloc[:, :i].copy()
model1 = LinearRegression()
model1.fit(X_tr,Y_train)
```

Figure 2.3 Linear Regression function

No additional arguments were included in the “LinearRegression” function, thus the arguments are in default mode as set by the source code in sklearn library.

Data transformation using box cox

Since boxcox uses logarithm function, the input data has data that are between 0 to 1 thus a constant number of 3 is added to every row of the data, that is shown below:

```
from scipy import stats
X_copy = X.copy()
No_of_variable = len(X_copy.columns)
#X.values.ravel()
# NEED TO CHANGE ALL SMALL NUMBERS (0 to 1)
Num_row = len(X_copy)

for i in range (Num_row):
    X_copy.iloc[i] = X_copy.iloc[i] + 3
X_copy
```

Figure 2.4 Adding of constant number 3 to every row

Dataset was transformed using “boxcox” function from scipy library in python shown below:

```
#Extract every column
Columns = ["fixed acidity","volatile acidity", "citric acid", "residual sugar",
           "chlorides", "free sulfur dioxide", "total sulfur dioxide", "density", "pH", "sulphates", "alcohol"]

X_boxdf = pd.DataFrame()


for i in range(No_of_variable):
    #extract the column


    new_df1 = X_copy.loc[:,Columns[i]]

    #flatten it to 1D
    A = new_df1.values.ravel()

    #boxcox it
    X_box1, fitted_lambda1 = stats.boxcox(A)

    print(fitted_lambda1)

    #put them into a dataframe
    df1 = pd.DataFrame (X_box1, columns = [Columns[i]])

    #append each column in a dataframe
    X_boxdf = pd.concat([X_boxdf,df1],axis =1)


display(X_boxdf)
```

Figure 2.5 Data transformation using boxcox

The ‘boxcox’ function could only process 1D data, thus each row of the input data was extracted and flatten.

Results before and after boxcox:

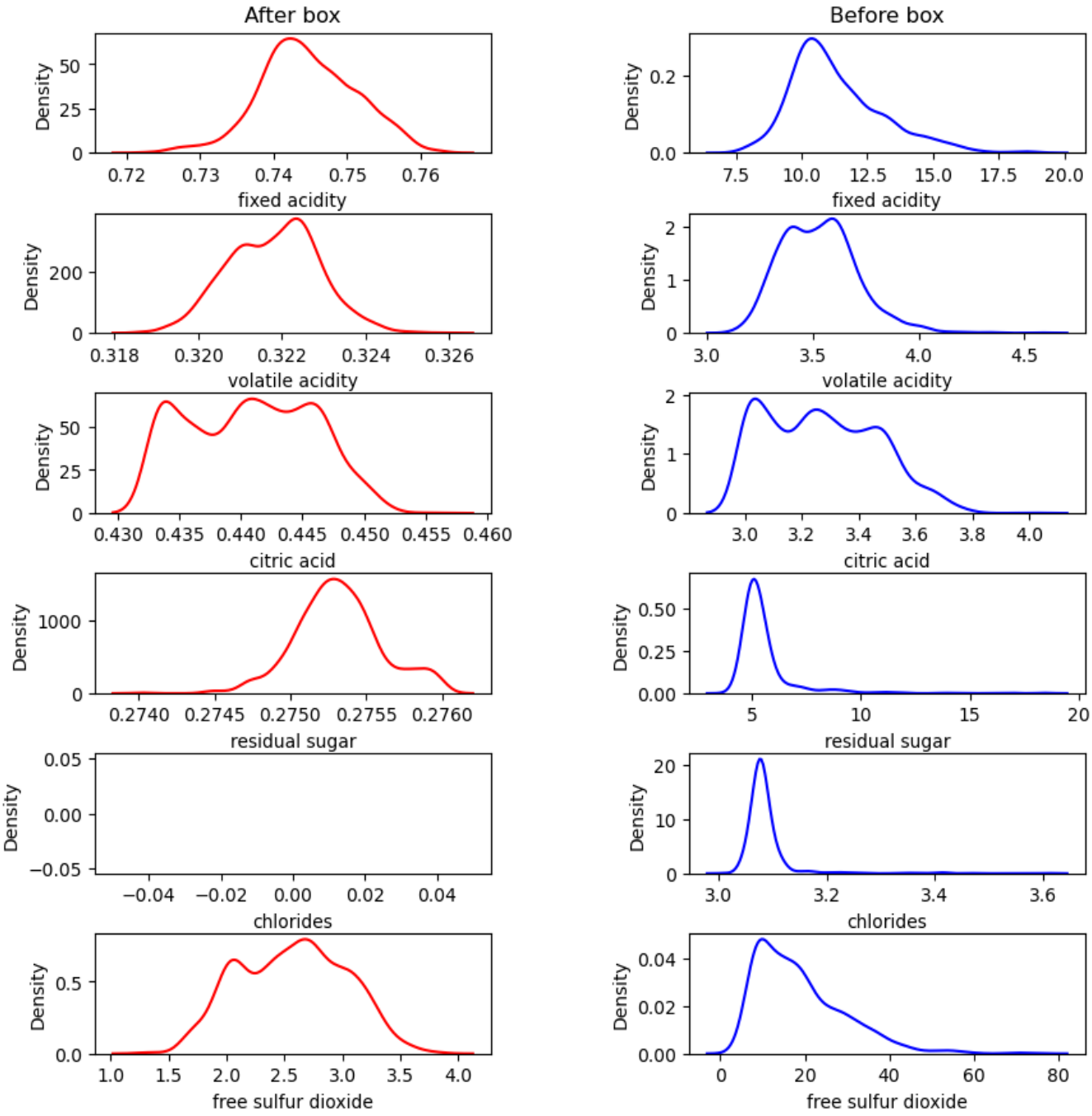


Figure 2.6 Part 1 of density plot of the input data using boxcox

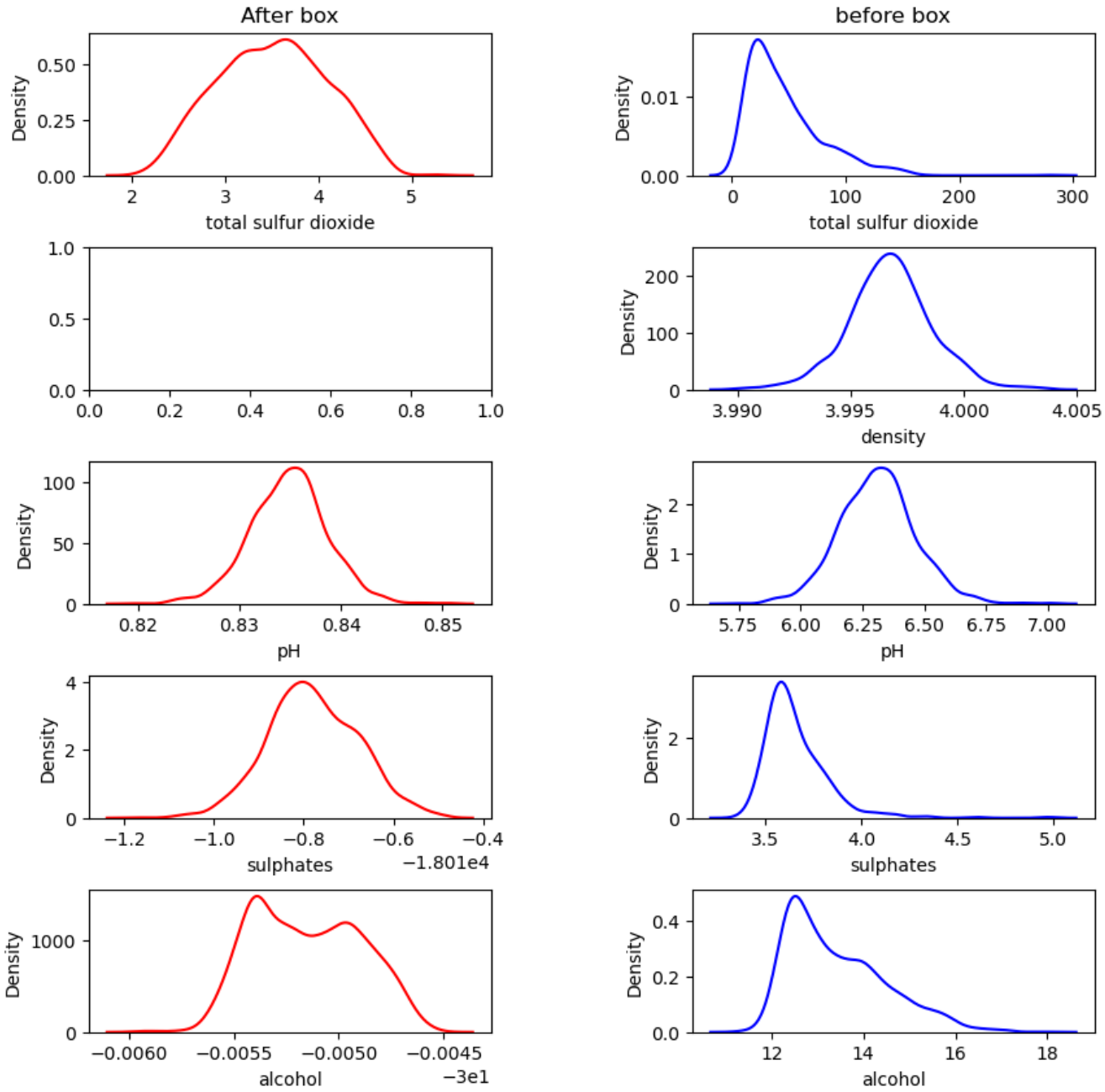


Figure 2.7 Part 2 of density plot of the input data using boxcox

From figure 2.6 & 2.7, it could be seen that the data has been successfully transformed except for chloride and density. Boxcox was unable to normalize chloride, this could be because of the large number of outliers and nature of range of chloride reading that is small. Density data naturally looks like a normally distributed data.

2.2 Gaussian Process Regression

Raw input data was used in this regression, as this regression has multiple kernels to explain the nature of the data and data inserted into this regression is automatically has an assumed mean of 0.

```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import DotProduct, WhiteKernel,RBF,
#https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpr_
''' Try out different kernels, adjust alpha'''
kernel1 = DotProduct() +WhiteKernel()+ RationalQuadratic()
kernel2 = WhiteKernel()+ RationalQuadratic()
kernel3 = WhiteKernel()+ DotProduct()

#adding two kernels can be thought of as an OR operation. That is, if you
model2 = GaussianProcessRegressor(kernel = kernel1)
model2.fit(X_train,Y_train.values.ravel())
print(model2.score(X_test,Y_test.values.ravel()))

```

Figure 2.8 Gaussian process regression modelling

From figure 2.8, regression is done by selecting the kernels that best explain the nature of the data with comparison on its different R^2 value, mean squared error and sum squared error. RBF is not tested as Rational Quadratic is a better version of it.

For all the kernels included, all the settings were in default:

1) **RationalQuadratic** (length_scale=1.0, alpha=1.0, length_scale_bounds=(1e-05, 100000.0), alpha_bounds=(1e-05, 100000.0))
2) **DotProduct**(sigma_0=1.0, sigma_0_bounds=(1e-05, 100000.0))
3) **WhiteKernel**(noise_level=1.0, noise_level_bounds=(1e-05, 100000.0))[source]

Various Kernel Combination tested:

Kernel 1: DotProduct() + WhiteKernel() + RationalQuadratic()

Kernel 2: WhiteKernel() + RationalQuadratic()

Kernel 3: WhiteKernel() + DotProduct()

3. Results & Discussion

3.1 Least Square Regression

	Untransformed Data	Transformed Data
R^2	0.3405	0.3443
Sum Squared Error (SSE)	265.94	264.42
Mean Squared Error (MSE)	0.4155	0.4132

Table 3.1 Comparison of untransformed and transformed test data R^2, SSE, MSE

Based on table 3.1, there are no significance difference for R^2, SSE, MSE between untransformed test data and transformed test data. This indicates that normalizing the data with boxcox function does not improve the regression model results much.

Variable added	R^2 train	R^2 test	Adj R^2 train	Adj R^2 test	Percentage Change (%)
Fixed Acidity	0.015095	0.015686	0.014065	0.014143	
Volatile Acidity	0.174847	0.115424	0.173120	0.112647	696.49
Citric Acid	0.175143	0.114645	0.172552	0.110469	-1.93
Residual Sugar	0.175277	0.114797	0.171819	0.109221	-1.13
Chlorides	0.186291	0.129599	0.182021	0.122735	12.37
Free Sulfur Dioxide	0.188238	0.133614	0.183122	0.125402	2.17
Total Sulfur Dioxide	0.218688	0.159971	0.212937	0.150667	20.15
Density	0.282924	0.209990	0.276885	0.199974	32.73
pH	0.289626	0.217210	0.282889	0.206028	3.03
Sulphates	0.333974	0.284147	0.326948	0.272767	32.39
Alcohol	0.370541	0.340477	0.363229	0.328925	20.59

Table 3.2 R^2 and Adjusted R^2 value of untransformed data

From table 3.2, it shows that volatile acidity has a significant impact on model prediction performance based on the percentage change for adjusted R^2 test of 696.49%. Chlorides, total sulfur dioxide, density,

sulphates and alcohol as well has substantial impact on the model prediction performance with a range of 12%-32% percentage change. Citric acid, residual sugar, free sulfur dioxide and pH even though it has negative or minute positive impact on the adjusted R^2, it was not considered to be taken off the model as adjusted R^2 only shows the variable has little statistically significance but not the significance in theory for red wine quality.

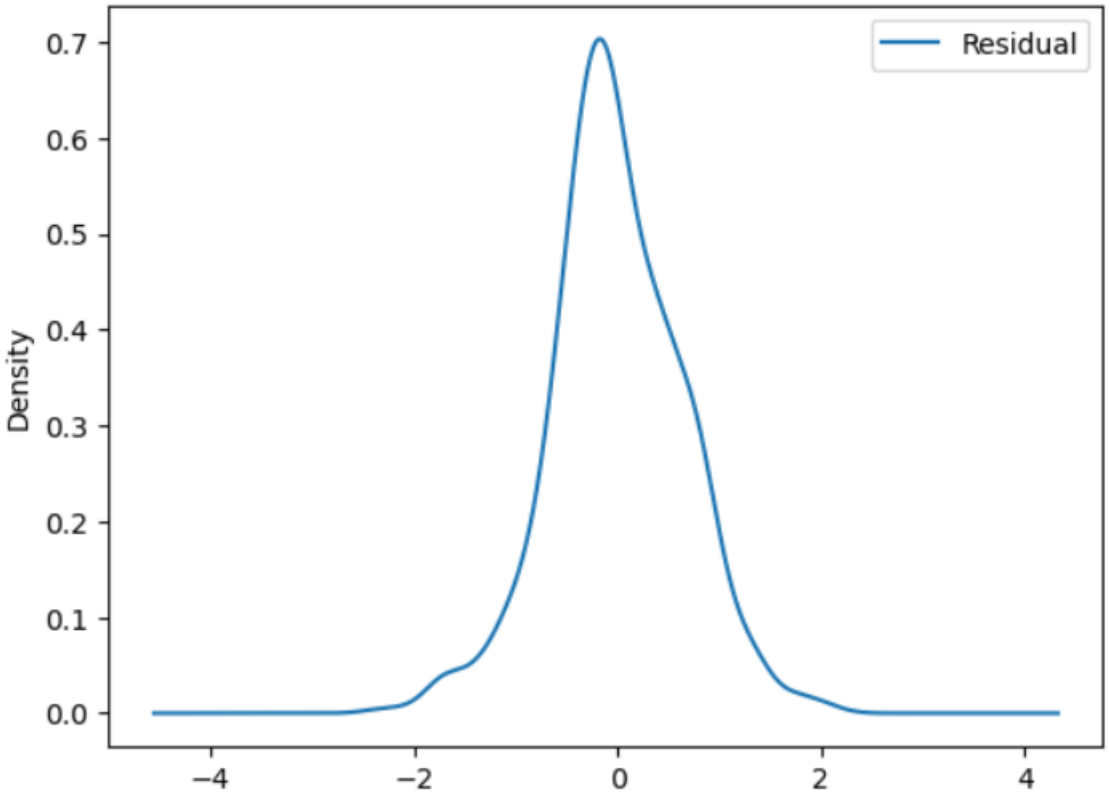


Figure 3.1 Residual analysis using kernel density estimation

In figure 3.1, it shows that the residuals resemblance a normally distributed density, with constant variance and a mean of 0. Thus, least square regression model is still appropriate to be used for prediction.

3.2 Gaussian Process Regression

	Kernel 1	Kernel 2	Kernel 3
R^2	0.4389	0.3032	0.3408
Sum Squared Error (SSE)	224.98	210.05	205.73
Mean Squared Error (MSE)	0.3515	0.3282	0.3215

Table 3.3 Comparison of between kernel 1,2 and 3 test data R^2, SSE, MSE

In table 3.3, it shows that kernel 1 has higher R^2, thus data variability is explained better with Kernel 1, but its SSE and MSE it's the highest when compared with kernel 2 and 3. Kernel 1 might not be the best combined kernel for this regression. Kernel 3 would be used as the chosen kernel, as its SSE and MSE is the lowest with a R^2 value that is in between kernel 1 and 2.

3.3 Comparing both models

	Linear least square	Gaussian Process Regression	Percentage difference (%)
R^2	0.3443	0.3408	-1.02
Sum Squared Error (SSE)	264.42	205.73	-22.20
Mean Squared Error (MSE)	0.4132	0.3215	-22.19

Table 3.4 Comparison of between kernel linear least square and gaussian process regression test data R^2, SSE, MSE

In table 3.4, gaussian process regression has a lower R^2 value of 1.02% when compared to linear least square. As for SSE and MSE, there is a reduction of 22.20% and 22.19% respectively when compared between the 2 models. Thus, this shows that gaussian process regression model is a better model to explain the red wine data set and provided better prediction confidence as well.

4. Conclusion

In conclusion, gaussian process regression is a better model to explain the variation of the red wine data and better accuracy in prediction as well. Winemakers and customers can attain an accurate prediction of the wine quality from its physicochemical properties. This reduces the need to do sensory test performed by human taste experts which are expensive, slow, and complex.

5. References

[1] O. Knagg, "An intuitive guide to Gaussian processes - Towards Data Science," *Medium*, Jan. 15, 2019. [Online]. Available: https://towardsdatascience.com/an-intuitive-guide-to-gaussian-processes-ec2f0b45c71d.

[2] "1.7. Gaussian Processes," *scikit-learn*, 2015. [Online]. Available: https://scikit-learn.org/stable/modules/gaussian_process.html#.

[3] "Kernel Cookbook," *Toronto.edu*, 2023. [Online]. Available: https://www.cs.toronto.edu/~duvenaud/cookbook/. [Accessed: Apr. 09, 2023]

Appendix

Linear least square regression

Non-transformed data

1)

```
#Data splitting first before scaling, as it prevents data leakage:
#https://machinelearningmastery.com/data-preparation-without-data-leakage/
X_train, X_test, Y_train, Y_test = train_test_split(X,Y,test_size = 0.4,random_state=1,shuffle = True)

print(X_train.info())
print(Y_train.info())
print(X_test.info())
print(Y_test.info())

```

2)

```
from sklearn.metrics import r2_score\
#https://builtin.com/data-science/adjusted-r-squared
#https://statisticsbyjim.com/regression/interpret-adjusted-r-squared-predicted-r-squared-regression/
result_df = pd.DataFrame()
for i in range(1, len(mutual_info) + 1):
    X_tr = X_train.iloc[:, :i].copy()
    X_te = X_test.iloc[:, :i].copy()
    model1 = LinearRegression()
    model1.fit(X_tr,Y_train)

    Prediction1 = model1.predict(X_tr)
    R1 = r2_score(y_true= Y_train,y_pred = Prediction1)
    Prediction2 = model1.predict(X_te)
    R2 = r2_score(y_true= Y_test,y_pred = Prediction2)
    adj_r2_train = 1 - ((1 - R1) * (len(X_tr) - 1) / (len(X_tr) - i - 1))
    adj_r2_test = 1 - ((1 - R2) * (len(X_te) - 1) / (len(X_te) - i - 1))

    result_df = result_df.append(pd.DataFrame({'r2_train': R1,'r2_test':R2,
                                               'adj_r2_train': adj_r2_train, 'adj_r2_test': adj_r2_test}, index=[i]))

result_df
```

Transformed data

1)

```
from scipy import stats
X_copy = X.copy()
No_of_variable = len(X_copy.columns)
#X.values.ravel()
# NEED TO CHANGE ALL SMALL NUMBERS (0 to 1)
Num_row = len(X_copy)

for i in range (Num_row):
    X_copy.iloc[i] = X_copy.iloc[i] + 3
X_copy
```

2)

```
#Extract every column
Columns = ["fixed acidity","volatile acidity", "citric acid", "residual sugar",
           "chlorides", "free sulfur dioxide", "total sulfur dioxide", "density", "pH", "sulphates", "alcohol"]

X_boxdf = pd.DataFrame()


for i in range(No_of_variable):
    #extract the column


    new_df1 = X_copy.loc[:,Columns[i]]

    #flatten it to 1D
    A = new_df1.values.ravel()

    #boxcox it
    X_box1, fitted_lambda1 = stats.boxcox(A)

    print(fitted_lambda1)

    #put them into a dataframe
    df1 = pd.DataFrame (X_box1, columns = [Columns[i]])

    #append each column in a dataframe
    X_boxdf = pd.concat([X_boxdf,df1],axis =1)


display(X_boxdf)
```

3)

```
#Current lamda for training Chlorides:-51.29765249530617
#Density: -10.549758693816631
#sulphates:-9.559528837307447
#alcohol:-4.2188919493230825

sulphates = X_boxdf[["sulphates"]]

    #flatten it to 1D
C = sulphates.values.ravel()
    #boxcox it
X_sulphates = stats.boxcox(C, lmbda = -5)

#print(X_sulphates )
df3 = pd.DataFrame (X_sulphates, columns = ['sulphates'])

alcohol = X_boxdf[["alcohol"]]
    #flatten it to 1D
d = alcohol.values.ravel()
    #boxcox it
X_alcohol = stats.boxcox(d, lmbda = -3)
#print(X_chloride)
df4 = pd.DataFrame (X_alcohol, columns = ['alcohol'])
A
df_train_box_various = pd.concat([df1,df3,df4],axis = 1)
display(df_train_box_various)

fig, axs = plt.subplots(3,2,figsize=(10,10))
plt.subplots_adjust(hspace=0.5)
plt.subplots_adjust(wspace=0.5)
sb.kdeplot(ax = axs[0,0], data=df1, x = 'chlorides', color = 'red' , label = 'After Box')
sb.kdeplot(ax = axs[0,1], data=X_train, x = 'chlorides',color = 'blue', label = 'Before Box')
axs[0, 0].set_title("After box")
axs[0, 1].set_title("before box")


sb.kdeplot(ax = axs[1,0], data=df3, x = 'sulphates', color = 'red' , label = 'After Box')
sb.kdeplot(ax = axs[1,1], data=X_train, x = 'sulphates',color = 'blue', label = 'Before Box')

sb.kdeplot(ax = axs[2,0], data=df4, x = 'alcohol', color = 'red' , label = 'After Box')
sb.kdeplot(ax = axs[2,1], data=X_train, x = 'alcohol',color = 'blue', label = 'Before Box')
```

4)

```
Various_column = ['sulphates','alcohol']


for i in range(2):
    #extract the column
    X_boxdf[Various_column[i]] = df_train_box_various[Various_column[i]]
    #X_test_boxdf[Various_column[i]] = df_test_box_various[Various_column[
#     new_df1 = X_train_box.loc[:,Various_column[i]]
#     new_df2 = X_test_box.loc[:,Various_column[i]]

X_boxdf['chlorides'] = X_copy['chlorides']
X_boxdf['density'] = X_copy['density']
#display(df_train_box_various)
display(X_boxdf)

# display(df_test_box_various)
# display(X_test_boxdf)

```

5)

```
X_train_box, X_test_box, Y_train, Y_test = train_test_split(X_boxdf,Y,test_size = 0.4,random_state=1,shuffle = True)
display(X_train_box)
```

6)

```
result_df1 = pd.DataFrame()
# X_tr1 = X_train_boxdf.iloc[:, :1]
# display(X_tr)
# model2 = LinearRegression()
# model2.fit(X_tr1,Y_train)
# Prediction3 = model2.predict(X_tr1)
# R1_1 = r2_score(y_true= Y_train,y_pred = Prediction2)
# display(R1_1)
for k in range(1,len(mutual_info1)+1):
    X_tr1 = X_train_box.iloc[:, :k]
    X_te1 = X_test_box.iloc[:, :k]
    model2 = LinearRegression()
    model2.fit(X_tr1,Y_train)

    Prediction3 = model2.predict(X_tr1)
    R1_1 = r2_score(y_true= Y_train,y_pred = Prediction3)
    Prediction4 = model2.predict(X_te1)
    R2_2 = r2_score(y_true= Y_test,y_pred = Prediction4)
    adj_r2_train_1 = 1 - ((1 - R1_1) * (len(X_tr1) - 1) / (len(X_tr1) - k - 1))
    adj_r2_test_1 = 1 - ((1 - R2_2) * (len(X_te1) - 1) / (len(X_te1) - k - 1))

    result_df1 = result_df1.append(pd.DataFrame({'r2_train': R1_1,'r2_test':R2_2,
                                            'adj_r2_train': adj_r2_train_1, 'adj_r2_test': adj_r2_test_1}, index=[k]))


result_df1
```

7)

```
model2 = LinearRegression()
model2.fit(X_train_box,Y_train)

Y_pred_box = model2.predict(X_test_box)
Residual1 = Y_test - Y_pred_box
print(Residual1)
plt.hist(Residual, bins = 30)
plt.show()

SSE1 = np.sum((Y_test - Y_pred_box)**2)
print(SSE1)

MSE1 = SSE1/len(Y_test)
print(MSE1)
```

Gaussian process regression

1)

```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import DotProduct, WhiteKernel,RBF, RationalQuadratic, ExpSineSquared
#https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpr_prior_posterior.html#sphx-glr-auto-examples-gaussia
''' Try out different kernels, adjust alpha'''
kernel1 = DotProduct() +WhiteKernel()+ RationalQuadratic()
kernel2 = WhiteKernel()+ RationalQuadratic()
kernel3 = WhiteKernel()+ DotProduct()
#adding two kernels can be thought of as an OR operation. That is, if you add together two kernels, then the resulting kerne
model2 = GaussianProcessRegressor(kernel = kernel1)
model2.fit(X_train,Y_train.values.ravel())
print(model2.score(X_test,Y_test.values.ravel()))

```

2)

```
Y_pred2 = model2.predict(X_test)
Y_pred2df = pd.DataFrame(Y_pred2,columns = ['quality'])

#print(Y_pred2)
#type(Y_testnp2)
SSE2 = np.sum((Y_test - Y_pred2df)**2)
print(SSE2)

MSE2 = SSE/len(Y_test)
print(MSE2)
```

```
quality    205.726061
dtype: float64
quality    0.321447
dtype: float64
```