COMP7703 - Machine Learning

Name: Jaskeerat Singh

Student ID: 47610039

This project will use various machine learning algorithms to apply classification and regression techniques to the Australian Food Composition Database data. The dataset consists of 2 sheets - Solid & Liquid and only Liquid. The Solid & Liquid sheet consists of 293 columns, and the Liquid sheet consists of 204 columns. The project's overall objective will be to check the basic statistics of the data, clean the data using pre-processing techniques and apply classification/regression techniques to the cleaned data.

Pre-Processing

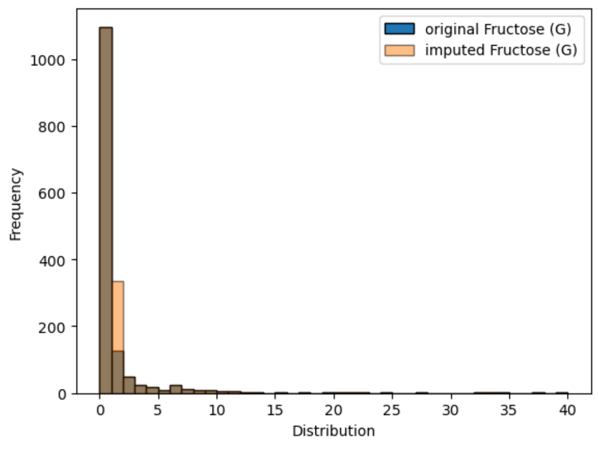
Removing Nulls

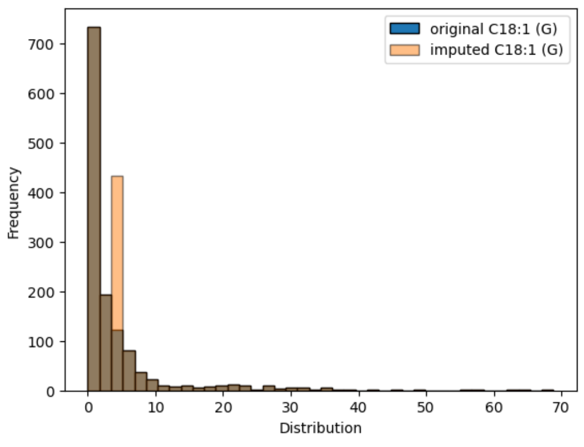
First, we will check the percentage of nulls in each column and drop columns having nulls greater than or equal to 60%. After dropping columns with a high percentage of nulls, we will check and drop any duplicate rows.

Citric Acid (G)	Fumaric Acid (G)	Lactic Acid (G)	Malic Acid (G)	Oxalic Acid (G)	Propionic Acid (G)	Quinic Acid (G)	Shikimic Acid (G)	Succinic Acid (G)	Tartaric Acid (G)	Aluminium (Al) (Ug)	Antimony (Sb) (Ug)	Arsenic (As) (Ug)	Cadmium (Cd) (Ug)
68.811881	99.938119	68.688119	69.059406	81.683168	91.212871	69.925743	99.814356	98.700495	97.462871	77.784653	90.594059	70.792079	73.267327
45.502646	100.000000	44.973545	45.502646	90.476190	79.894180	47.089947	100.000000	89.417989	85.185185	51.322751	94.708995	51.851852	56.613757

Imputation

After dropping columns, we will get data containing a low percentage of nulls. To impute and fill null values, Imputation will be carried out to remove the nulls and blank values. Here, we will use the mean of the column for numerical columns and the mode for string columns.





Normalization

After the clean and imputed data has been created, we will normalize our data. Looking at the summary statistics of the data, different features of the dataset lie between different numerical ranges. Some features lie between the data ranges of [0 to 20], while others lie between the data ranges of [3500 to 0].

	Classification	Energy With Dietary Fibre, Equated (Kj)	Energy, Without Dietary Fibre, Equated (Kj)	Moisture (Water) (G)	Protein (G)	Nitrogen (G)	Fat, Total (G)	Ash (G)	Total Dietary Fibre (G)	Alcohol (G)
count	189.000000	189.000000	189.000000	189.000000	189.000000	189.000000	189.000000	189.000000	189.000000	189.000000
mean	19348.089947	673.804233	670.248677	76.659788	1.638095	0.264180	11.551323	1.029101	0.442328	1.344974
std	6438.857942	963.025866	962.193691	29.595268	2.703057	0.432222	26.910365	2.579306	1.443034	3.837350
min	20.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	14401.000000	159.000000	158.000000	74.000000	0.100000	0.020000	0.000000	0.100000	0.000000	0.000000
50%	19305.000000	293.000000	293.000000	88.300000	0.400000	0.060000	0.300000	0.300000	0.000000	0.000000
75%	23107.000000	574.000000	571.000000	93.700000	2.000000	0.320000	3.800000	0.900000	0.200000	0.000000
max	32102.000000	3404.000000	3404.000000	112.900000	20.400000	3.260000	92.000000	25.200000	13.500000	29.500000

To normalize the data, we will use min-max normalization. Min-max normalization is used for feature scaling to transform data within the specific range between 0 and 1. The normalized value follows the form:

X_normalized = (X - X_min) / (X_max - X_min)

- X will be the value in the feature
- X_min will be the minimum value in the particular feature
- X_max will be the maximum value in the particular feature
- The value will finally be normalized between 0 and 1

Feature Engineering

We will create features called "Type" in both Solid & Liquid and Liquid data frames. Here, Type will signify the form of food, like Solid or Liquid, where value = 1 will signify the row entry being liquid while 0 signifies the food entry being solid.

```
liquid = list(l_df["Classification"])

sl_df["Type"] = None
l_df["Type"] = None

l_df.loc[l_df["Classification"].isin(liquid), "Type"] = 1
sl_df.loc[sl_df["Classification"].isin(liquid), "Type"] = 1
sl_df.loc[~sl_df["Classification"].isin(liquid), "Type"] = 0
```

We will finally concat the two datasets to form a total dataframe. This dataframe will contain the columns common to both the solid/liquid and liquid dataframe. We will finally remove all the duplicate rows and form the final dataset. The final processed dataframe will consist of 1803 rows and 73 columns.

```python
total_df = pd.concat([sl_df, l_df], axis = 0)
total_df.drop_duplicates(inplace = True)
total_df = total_df.transpose().drop_duplicates().transpose()
total_df.drop(columns = string_columns, inplace = True)
total_df.dropna(inplace = True)
total_df = total_df.reset_index(drop = True)
```

Main Task

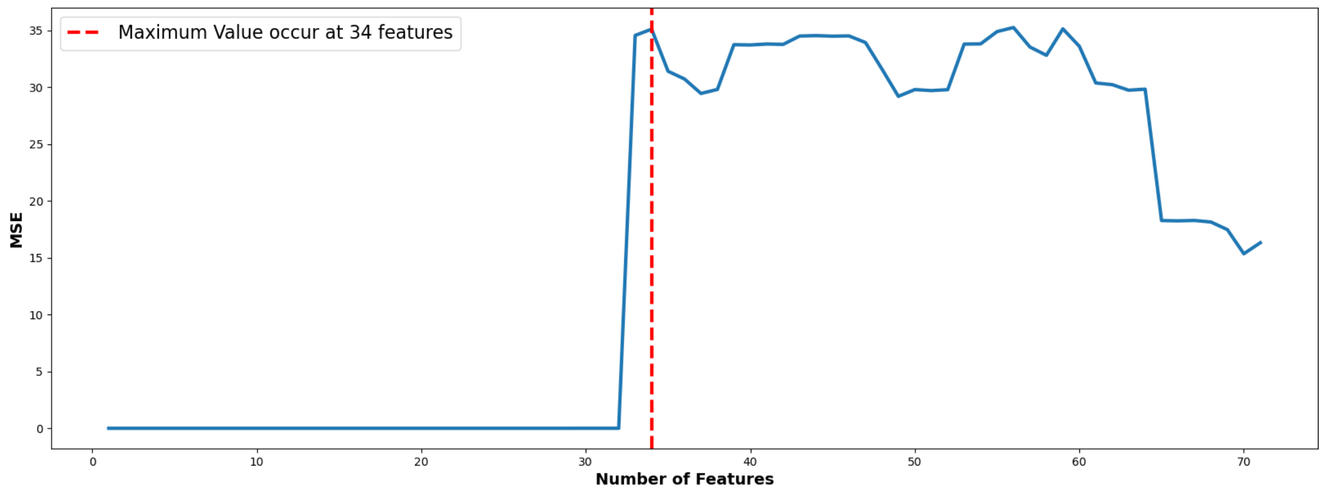
Supervised and unsupervised techniques are two primary types of machine learning algorithms. Supervised machine learning uses labelled data where a list of features corresponds to a final output label. The supervised task consists of classification and regression tasks –

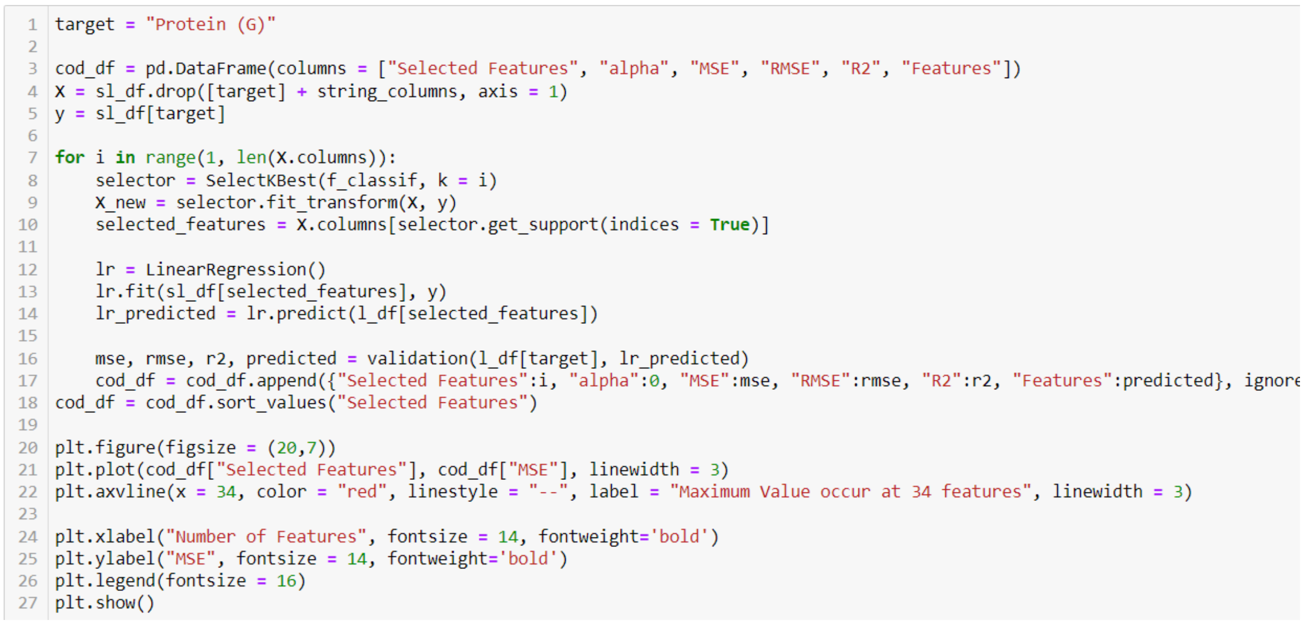
Classification consists of detecting and choosing a label from the training data's unique set of predefined labels.

Regression helps predict a continuous numerical value based on the values present in the training data. This value will be chosen from the range of values in the target column.

As our final data consists of many features, we will look into the problems of the curse of dimensionality arising from high-dimensional data. We will use the sklearn SelectKBest method for selecting the defined k features from a given dataset. It defines the relationship between the selected k features and the target feature for which it is being defined. SelectKBest help find the top K features. It helps identify the features which improve the model's features while reducing the dimensionality and improving model performance.

To check for the curse of dimensionality, we will plot the linear regression model where the line plot is plotted between an increasing number of features and the MSE calculated while predicting the Target value. For the prediction, train_test_split was performed to get train and test rows. The top K features were selected from the train data, and the values were predicted for test data.





```
target = "Protein (G)"

cod_df = pd.DataFrame(columns = ["Selected Features", "alpha", "MSE", "RMSE", "R2", "Features"])
X = sl_df.drop([target] + string_columns, axis = 1)
y = sl_df[target]

for i in range(1, len(X.columns)):
    selector = SelectKBest(f_classif, k = i)
    X_new = selector.fit_transform(X, y)
    selected_features = X.columns[selector.get_support(indices = True)]

    lr = LinearRegression()
    lr.fit(sl_df[selected_features], y)
    lr_predicted = lr.predict(l_df[selected_features])

    mse, rmse, r2, predicted = validation(l_df[target], lr_predicted)
    cod_df = cod_df.append({"Selected Features":i, "alpha":0, "MSE":mse, "RMSE":rmse, "R2":r2, "Features":predicted}, ignore
cod_df = cod_df.sort_values("Selected Features")

plt.figure(figsize = (20,7))
plt.plot(cod_df["Selected Features"], cod_df["MSE"], linewidth = 3)
plt.axvline(x = 34, color = "red", linestyle = "--", label = "Maximum Value occur at 34 features", linewidth = 3)

plt.xlabel("Number of Features", fontsize = 14, fontweight='bold')
plt.ylabel("MSE", fontsize = 14, fontweight='bold')
plt.legend(fontsize = 16)
plt.show()
```

From the graph we can say that as the number of features increase the value of MSE increases. After a while it reaches a high threshold and reduces back to normal. This sudden increase can be attributed to the overfitting of the data.

MSE or Mean Squared Error is defined as the average of the sum of squared difference between the predicted and actual values. MSE is defined by the formula as:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

For our regression Analysis, we will be using MSE to gauge the performance of different Models. The MSE is obtained by taking the average of the square of the predicted value and the original value. A low value of MSE indicates that the predicted value is closer to the original value. Hence, leading to better accuracy of the model.

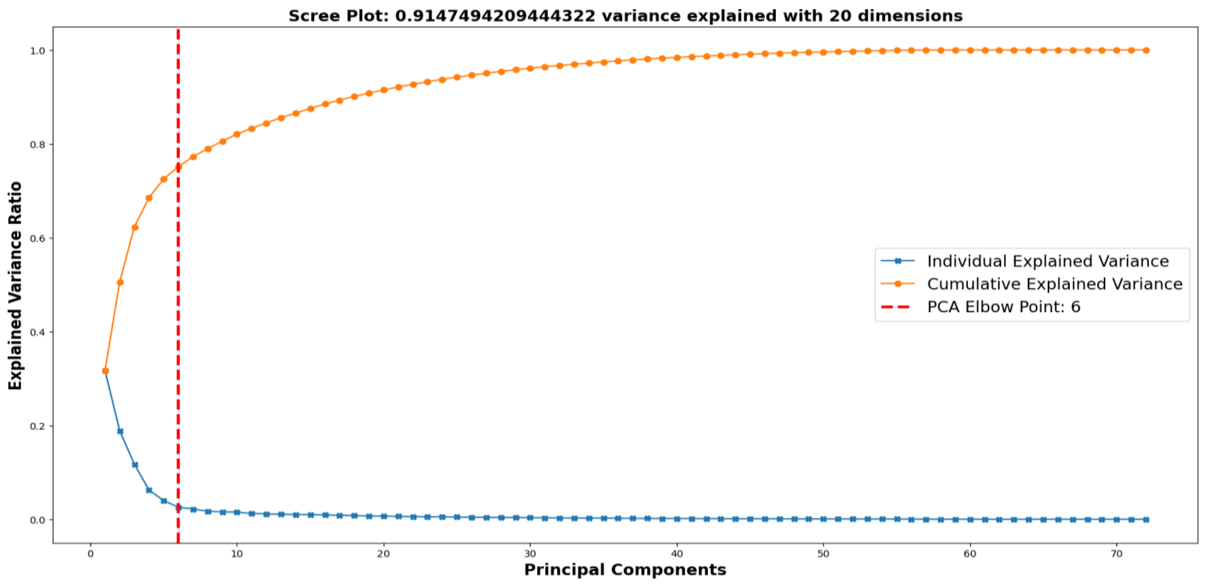
PCA

Principal component analysis, or PCA, is a technique used for dimensionality reduction. It helps us choose features from a dataframe by selecting the features which will not lose the information of trends and patterns from the dataset. PCA helps us choose features that do not reduce the accuracy of the prediction model. We will plot a scree plot for the total dataframe formed.

Method to calculate PCA :

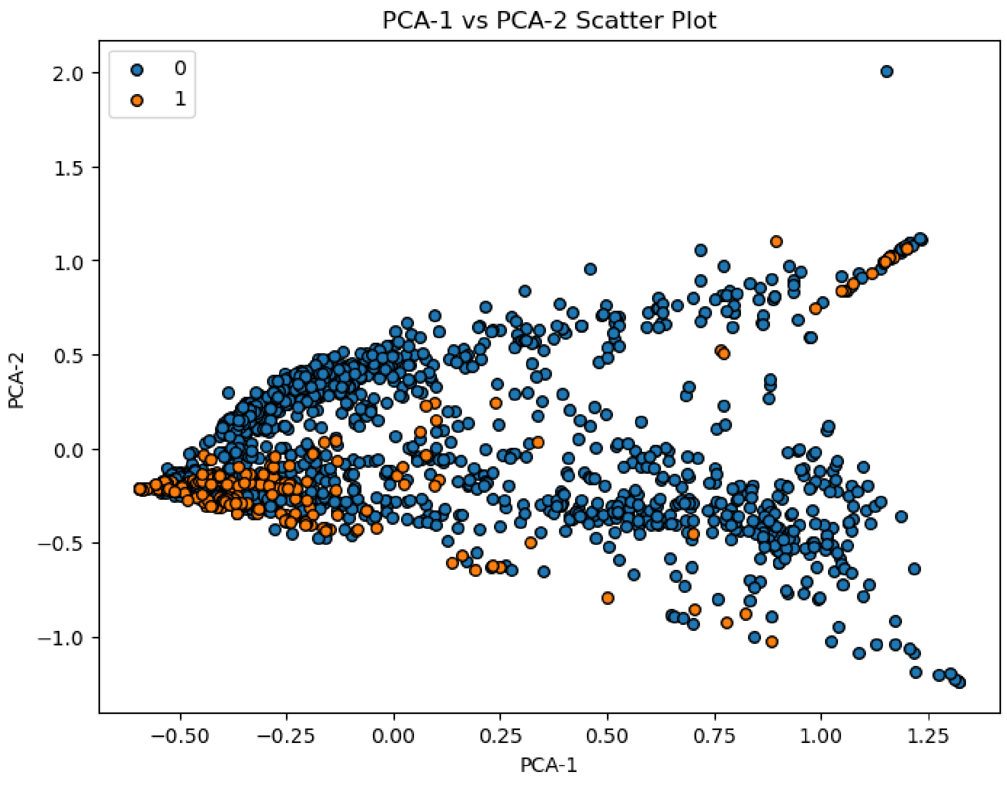
- Z-score normalisation of the data
- Computing the covariance matrix. Covariance matrix defines the relationship between the variables of the matrix.

- Compute eigenvalues and eigenvectors
- Selecting number of principal components based on elbow point



From the scree plot, we can see the elbow point at 6. It is defined as the point where the eigenvalue starts to level off. The eigenvalue is the variance brought in by adding a new principal component. The Plot shows that by adding more than 6 principal components, the variance does not significantly change for the model.

Based on the top 6 principal components, approximately 80% of the data's variance can be explained. Moreover, 92% of the variance can be explained with 20 dimensions.



If we plot the scatter plot of all the labels for the first 2 PCA values around the standard deviation, we can see that though there is some overlap, we can still see some clear divisions in the point values of the labels on the graph. For values with label 1, they are concentrated in one part of the graph while the other is distributed through the graph.

Based on the values found here, we will prove it using classification and regression models developed using the dataset.

Classification

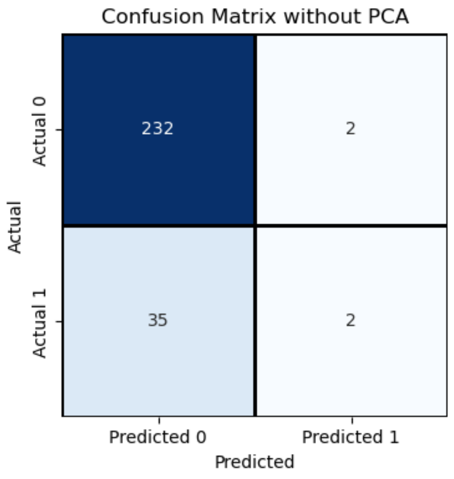
Classification is defined as a supervised machine learning task that helps in predicting the target label column based on the values already present in the dataset. The values of the dependent variable are computed based on the value of the independent variable. The objective of classification is to train and accurately predict label values of the previously unseen/unlabelled data.

SVM Classification

SVM classifier is a machine learning classifying algorithm that separates the data points of different labels(Target) using the hyperplane with a high margin. It means that the model tries to maximise the distance of each point from the hyperplane as much as possible without reducing the ability of the model to classify the data point correctly. SVM classifiers are known to be able to handle high-dimensional features.

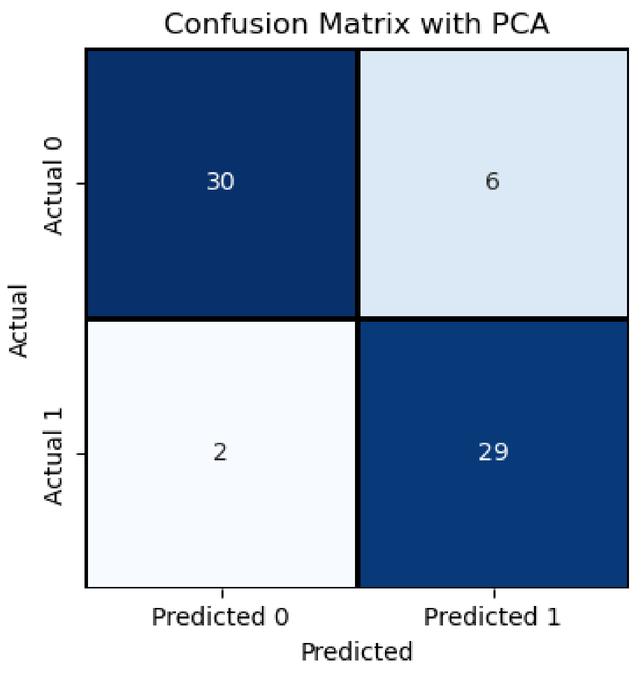
We will use the linear kernel to train the support vector classifier model. For training the model, we first use the train test split method to divide the dataset into 85 / 15 split. Here, we will train the model without using PCA first. After training the model, we will print the confusion matrix. For the development of the model, we have used the "Type" label column as the Target parameter. At the same time, all the other available numerical features act as the input for the independent variable.

The confusion matrix is defined as a table that describes the performance of the classification model by comparing the predicted value with the assigned label value. It does the same for every unique individual label value. It is defined chiefly for binary classification problems.

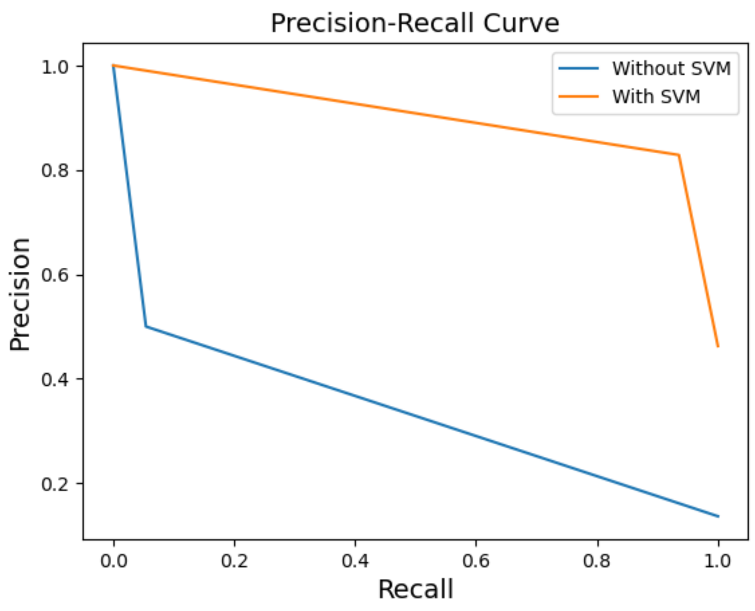


Without using PCA, we can see that the accuracy comes out to be 86%, but the F1 score comes out to be 10%.

When applying the same SVM method with PCA with a value of parameter component set to 20, as discussed before, we can see the accuracy increases to 88%, and the F1 value increases to 87%.



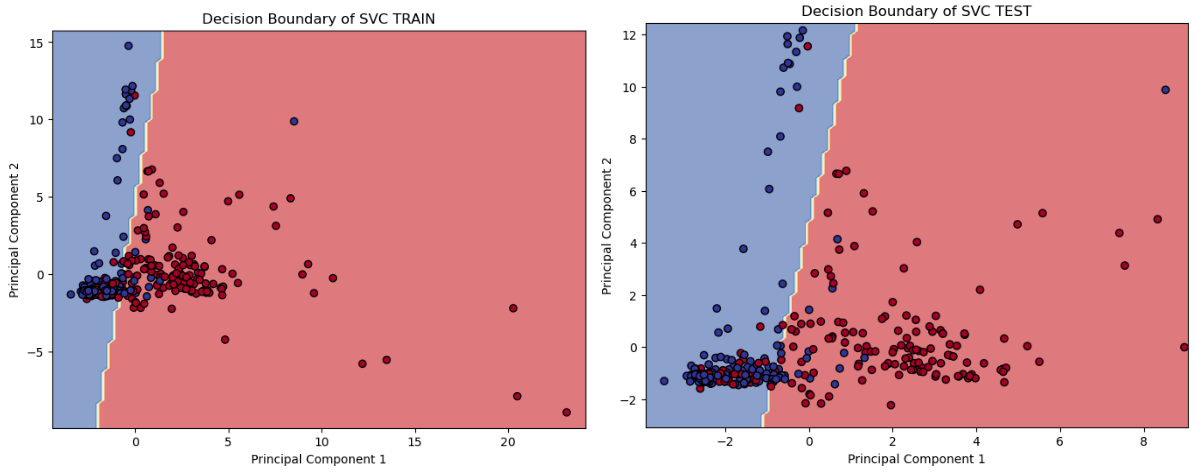
Without PCA, we can see that the model has high accuracy but fails in measuring a high F1 score because the model predicts most values as 0. With PCA, we can see the value of F1 increases significantly. The value of incorrect predictions of label 0 minutely but the value of correct prediction of label 1 increases significantly.



The precision and recall curve defines the graphical representation that compares the precision and recall of a binary classification model. It is a basic AUC-PC curve with a larger area under the curve means better performance.

The precision is defined as the ratio of true positives to the sum of true positives and false positives. It measures the accuracy of positive predictions.

Recall is defined as the ratio of true positives to the sum of true positives and false negatives. It checks how many positive instances were correctly identified.



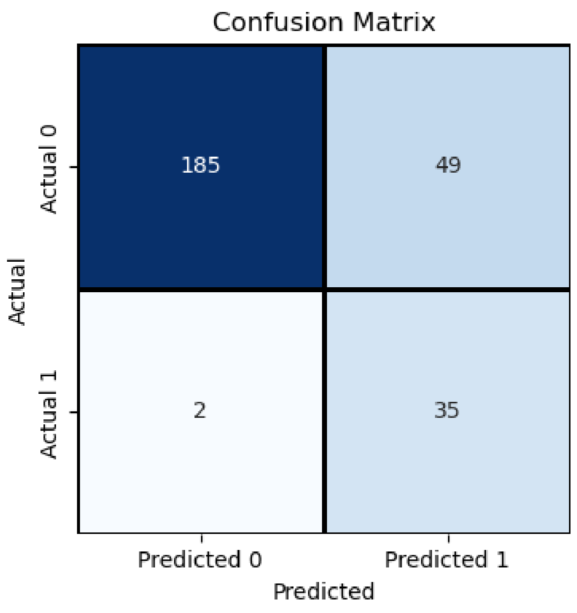
The implemented SVM Classifier with PCA works with PCA managing to displace the values of datapoints as far as possible from the defined hyperplane. The model works by clearly dividing the label values into 2 different portions.

Logistic Regression

Logistic regression is a binary classification model, even though it's a named regression. The goal of the trained model is to consistently score the data points to one of the outputs defined. In a typical example, this output or label can be typically represented as 0 and 1. The user can set the value of the probability as 0.6 out of a possible 1. This probability depends on the value of the other independent features present in the dataset. Based on this, the data points with a score of 0.6 and above are assigned as 1 else 0.

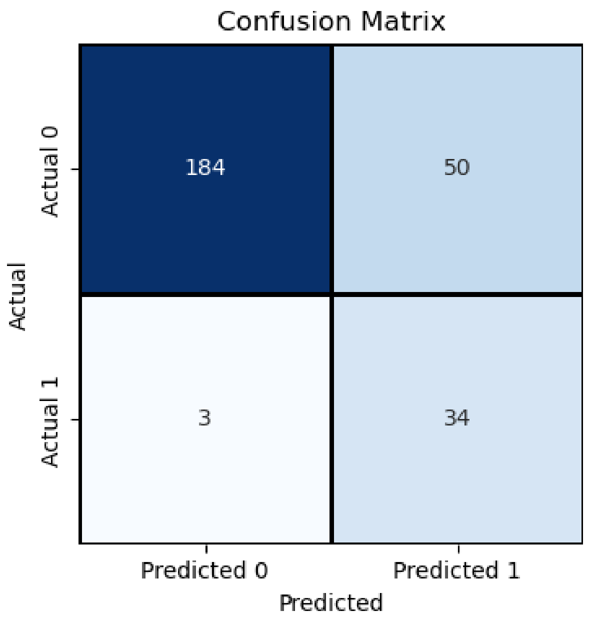
In our Analysis, we have performed logistic regression for the "Type" dependent variable. Train_test_split, like in SVM, was performed to train the model and use the test to get values of accuracy and f1. Without using cross-validation and PCA:

```
cv = StratifiedKFold(n_splits = 10)
cv_scores = cross_val_score(lr_model, x_test, y_test, cv = cv, scoring = "accuracy")
mean_accuracy = cv_scores.mean()
```



```
Accuracy using train_test_split:
0.8118081180811808

F1 using train_test_split:
0.578512396694215

Mean Accuracy using cross-validat
ion: 0.775
```

After performing PCA, with the number of components set to 20, there is a small decrease in the accuracy and F1 score. However, there is a significant increase in the value of accuracy using Cross-Validation Score. From the confusion matrix, we see that the number of incorrect predictions increases by 2, but the overall model performs better.

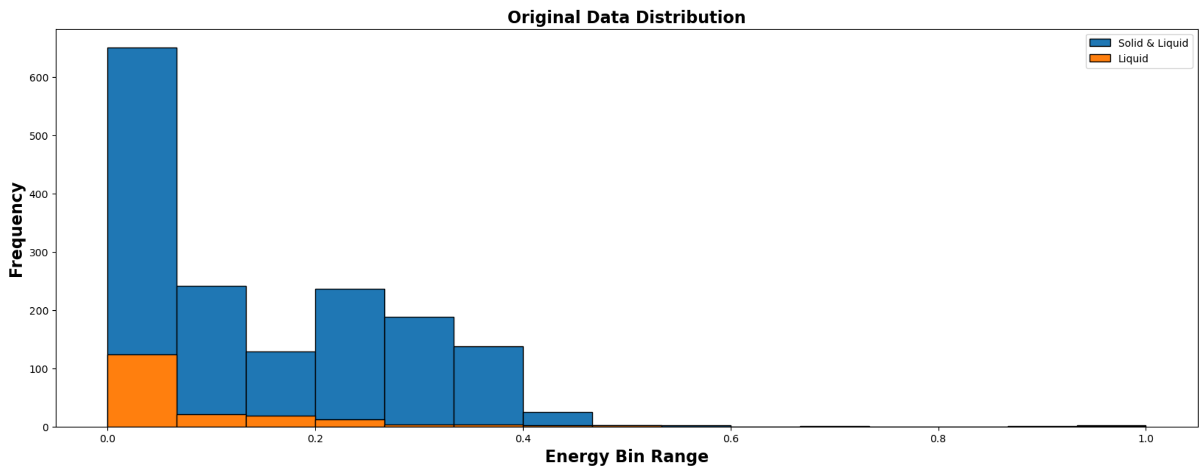


```
Accuracy using train_test_split with
PCA: 0.8044280442804428

F1 using train_test_split with PCA:
0.5619834710743802

Mean Accuracy using cross-validation:
0.8267195767195767
```

Overall, from the 2 models, SVM and Logistic Regression used for classification, we can say that SVM performs better by a better F1 score while maintaining decently similar accuracy. Linear regression works decently with the available data but fails to reduce a sizeable high-dimensional dataset.

Regression

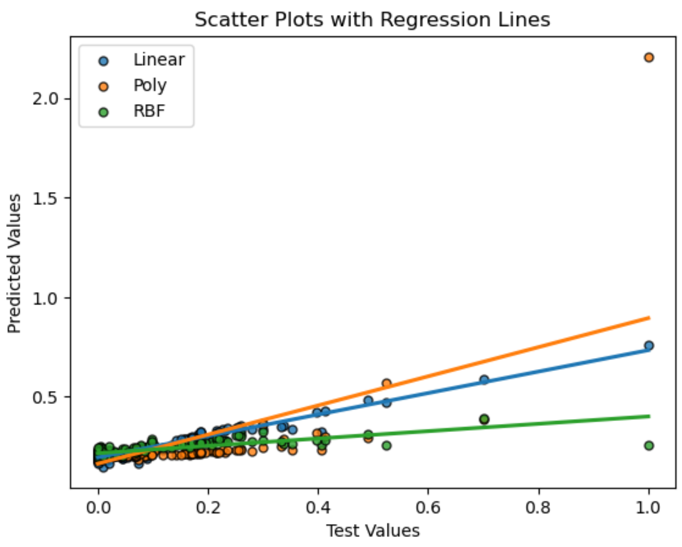
For the Regression tasks, we will be using the Solid & Liquid as the training dataset, while only Liquid values will be used as the testing dataset. For the Regression tasks, we have chosen the value of "**Protein (G)"** as the Target Column. The original "Protein (G)" column contains zero nulls in both the Solid & Liquid and Liquid dataset. It has a similar kind of distribution and lies under the same statistical variance bins.



SVR regression

Support vector regression is a part of the support vector machine (SVM) applied to regression problems. Here we will apply the regression model to predict the value of "Protein" using other present numerical independent features as input. Now we will compare the value of MSE and decide the best-performing kernel from Linear Poly and RBF.

Predicting the MSE values based on the default values of the model gives the Mean Squared Error as



```
Linear Mean Squared Error:
0.029310455927887925

Poly Mean Squared Error:
0.0345621818615044

RBF Mean Squared Error:
0.035048448419474985
```

```python
from sklearn.svm import SVR

target = "Protein (G)"
x_train = sl_df.drop(string_columns + [target], axis = 1).values
y_train = sl_df[target]
x_test = l_df.drop(string_columns + [target], axis = 1).values
y_test = l_df[target]
```

```python
svr_linear = SVR(kernel='linear', C=1.0, epsilon=0.2)
svr_poly = SVR(kernel='poly', degree=3, C=1.0, epsilon=0.2)
svr_rbf = SVR(kernel='rbf', C=1.0, epsilon=0.2)

svr_linear.fit(x_train, y_train)
svr_poly.fit(x_train, y_train)
svr_rbf.fit(x_train, y_train)

y_linear = svr_linear.predict(x_test)
y_poly = svr_poly.predict(x_test)
y_rbf = svr_rbf.predict(x_test)
```

```python
linear_mse = mean_squared_error(y_test, y_linear)
print("Linear Mean Squared Error:", linear_mse)
poly_mse = mean_squared_error(y_test, y_poly)
print("Poly Mean Squared Error:", poly_mse)
rbf_mse = mean_squared_error(y_test, y_rbf)
print("RBF Mean Squared Error:", rbf_mse)
```

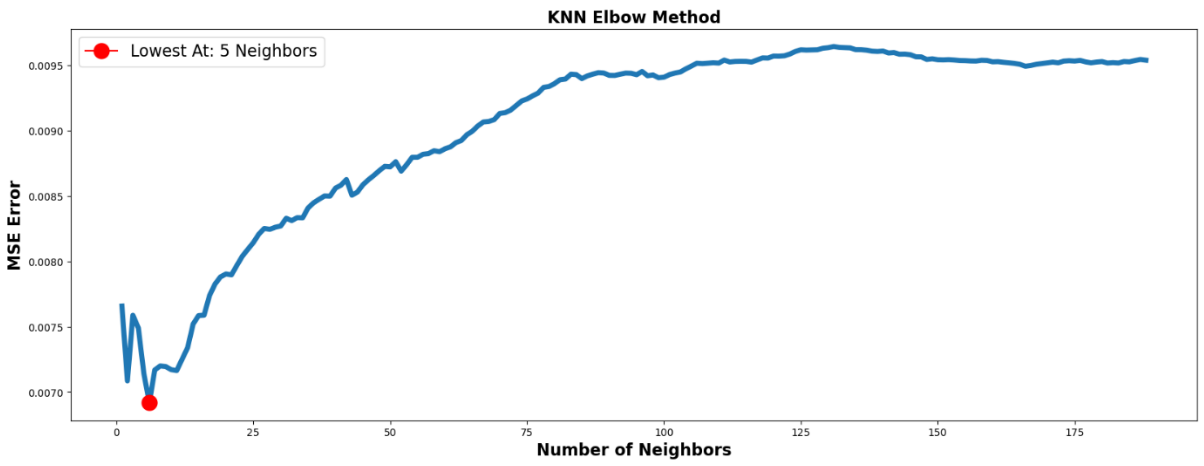
Knn

KNN regression is a supervised machine learning algorithm used for predicting the value of the target column, which in this case will be taken as "Protein". The value is predicted by considering the value of its K neighbouring data nodes. It is a simple program that works majorly on the basis of 1 hyperparameter K.

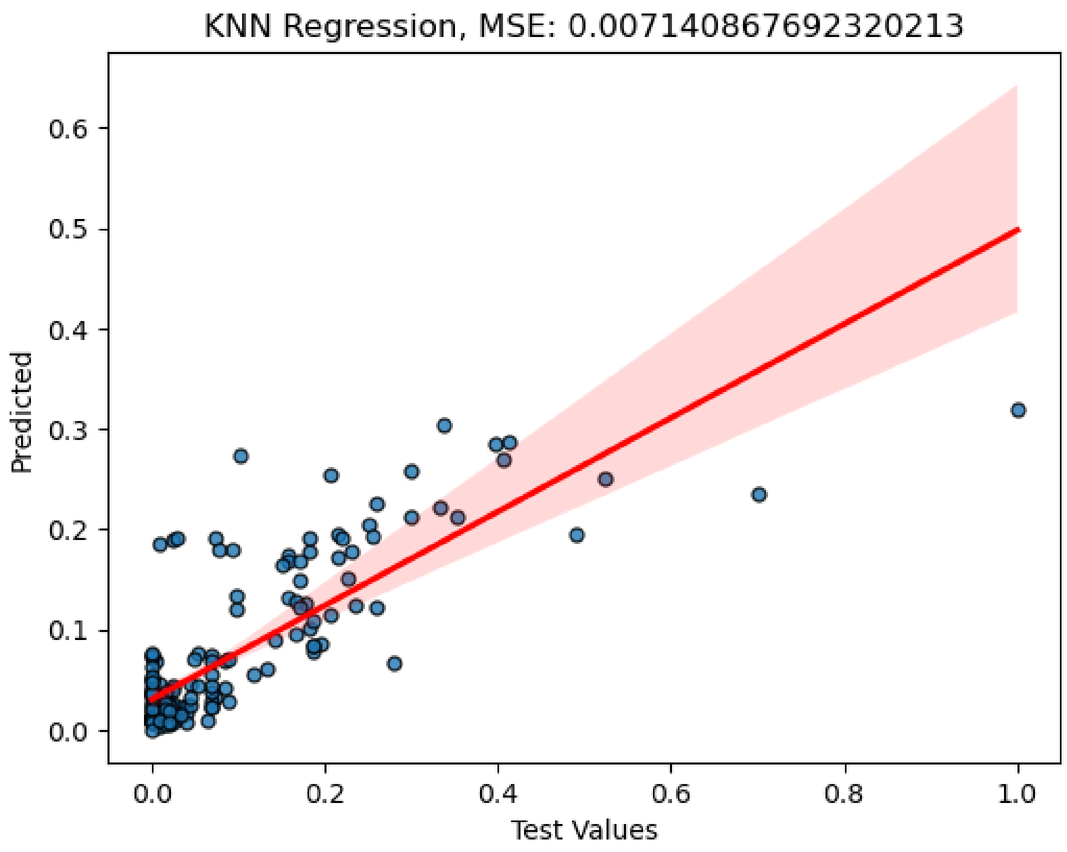
- KNN regression works by looking at the distance between the node to the K nearest neighbours.
- Take the average of these distances.
- Assigning the average value as the predicted value of a node.

To predict the best value of the hyperparameter K, we will use the elbow method. The elbow method is a plot that is plotted between MSE and the Number of neighbours.it is a sharp turn in the graph that resembles a human elbow.

```python
target = "Protein (G)"

error_rate = []
for i in range(1, l_df.shape[0]):
    knn = KNeighborsRegressor(n_neighbors = i)
    knn.fit(sl_df.drop([target] + string_columns, axis = 1), sl_df[target])
    knn_predicted = knn.predict(l_df.drop([target] + string_columns, axis = 1))
    mse = mean_squared_error(l_df[target], knn_predicted)
    error_rate.append(mse)
```



For the very low value of K, we can see that the model tries to overfit the data hence the value of MSE increases. On the other hand, a very small value of K under fits the data, which again results in a higher MSE value. From the plotted graph, we can see that the lowest value of MSE is observed when K = 5.



This plot helps visualize the relationship between the tested and predicted values. By calculating the MSE value, we can see that an MSE value of 0.007 suggests that the model works extremely well in predicting the value of "Protein" as close to the original value.

Comparing the MSE values shows that the KNN regressor works the best by producing the lowest MSE value.

Linear Regression

Linear regression is a model used to define the relationship between the independent features of the dataset and the dependent feature by fitting a line on this data. The algorithm assumes a linear relationship between the dependant and independent features. The line, depending on the number of features, can be represented by the formula:
$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p$

Where x_1, x_2, ..., x_p are the independent variables, β_0 is the y-intercept, and β_1, β_2, ..., β_p are the coefficients or slopes corresponding to each independent variable.
The overall objective of the model will be to reduce the squared error difference between predicted and observed values. Linear regression has two other forms depending upon how they handle multicollinearity. Multicollinearity occurs when different features are highly correlated to each other.
Linear regression: fits a line that reduces the sum of squared difference. It does not add any penalty for any coefficients, which can lead to overfitting.
Ridge Regression: Also known as L2 regularization, addresses the problem of multicollinearity or overfitting by introducing a regularization term alpha which helps in shrinking the coefficients to 0. L2 is the square root of the sum of squared coefficients.

$$\sum_{j=1}^{m}\left(Y_i - W_0 - \sum_{i=1}^{n} W_i X_{ji}\right)^2 + \alpha \sum_{i=1}^{n} W_i^2 = loss\_function + \alpha \sum_{i=1}^{n} W_i^2$$

Lasso Regression: Also known as L1 regularization, addresses the problem of multicollinearity or overfitting by introducing a regularization term alpha which reduces the coefficients to 0. This L1 is the sum of the absolute values of the coefficients. L1 is a form of feature selection that reduces the dependence on features in making the decision.

$$\sum_{j=1}^{m}\left(Y_i - W_0 - \sum_{i=1}^{n} W_i X_{ji}\right)^2 + \alpha \sum_{i=1}^{n} |W_i| = loss\_function + \alpha \sum_{i=1}^{n} |W_i|$$

Based on these, we will calculate and predict the value of “Protein” column.

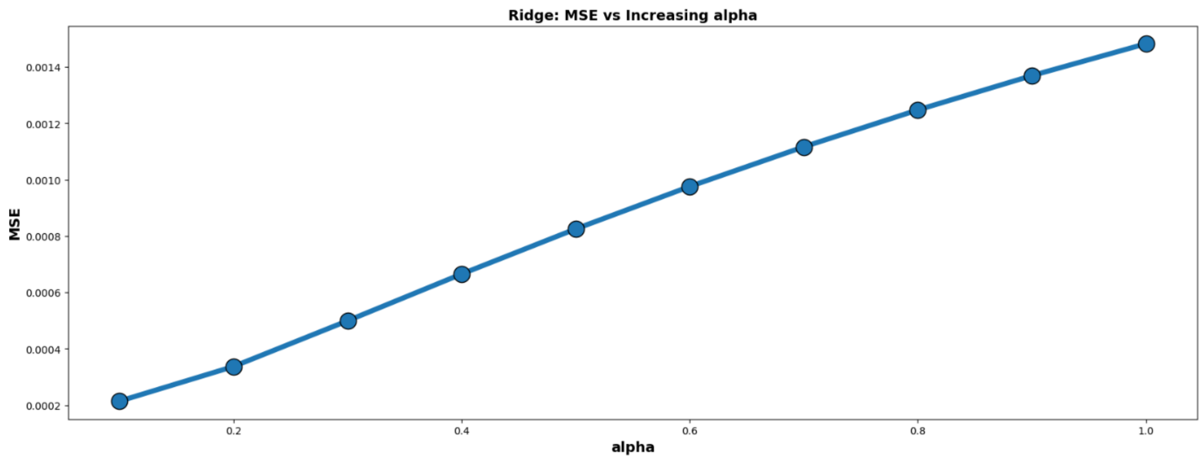
LR: MSE: 16.31638847345561 **RMSE:** 4.039354957596029 **R2:** -933.2815306801056

Ridge: MSE: 0.0012479646378167702 **RMSE:** 0.03532654296441658 **R2:** 0.9285411527219442

Lasso: MSE: 0.021837120222825822 **RMSE**: 0.14777388207266473 **R2:** -0.2504003652902915

Ridge Regression

```python
def validation(actual, predicted):
    mse = np.mean((actual - predicted) ** 2)
    rmse = np.sqrt(mse)
    r2 = 1 - (np.sum((actual - predicted) ** 2) / np.sum((actual - np.mean(actual)) ** 2))

    return mse, rmse, r2, predicted

target = "Protein (G)"

ridge_1_df = pd.DataFrame(columns=["alpha", "MSE", "RMSE", "R2"])
X = sl_df.drop([target] + string_columns, axis=1)
y = sl_df[target]

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

pca = PCA(n_components = 6)
X_pca = pca.fit_transform(X_scaled)

for j in np.arange(0.1, 1.1, 0.1):
    ridge_model = Ridge(alpha=round(j, 1))
    ridge_model.fit(X_pca, y)
    ridge_predicted = ridge_model.predict(pca.transform(scaler.transform(l_df.drop([target] + string_columns, axis=1))))

    mse, rmse, r2, predicted = validation(l_df[target], ridge_predicted)
    ridge_1_df = ridge_1_df.append({"alpha": round(j, 1), "MSE": mse, "RMSE": rmse, "R2": r2}, ignore_index=True)
    print("alpha:", round(j, 1), "Coefficients:", ridge_model.coef_, "MSE:", mse)

ridge_1_df = ridge_1_df.sort_values(by=["R2", "MSE"], ascending=[False, True])
```

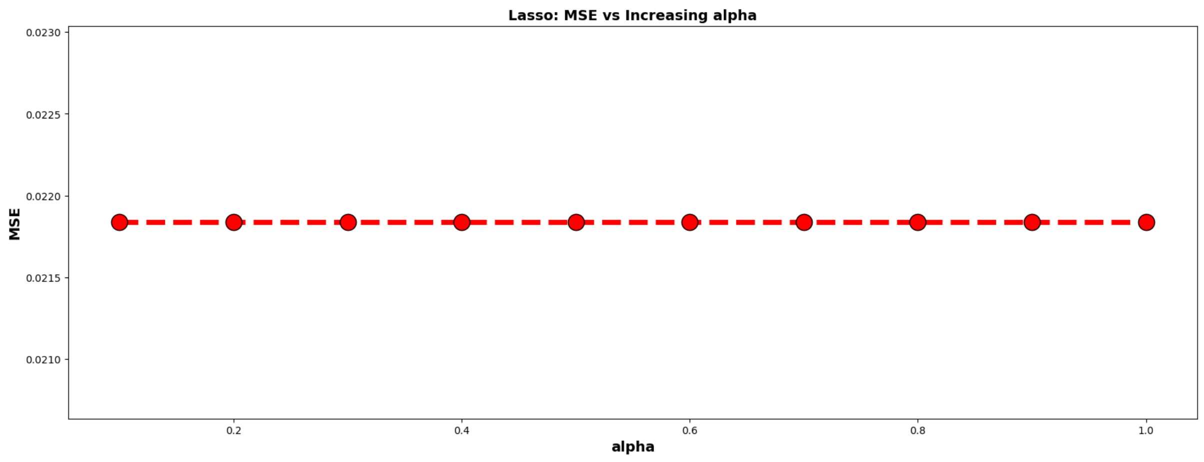


```
alpha: 0.1 Coefficients: [ 0.03040829 -0.01865157 -0.00095398 -0.01931942 -0.02083847  0.01063381] MSE: 0.012778336880218 64
alpha: 0.2 Coefficients: [ 0.03040808 -0.01865142 -0.00095397 -0.01931913 -0.0208381   0.01063361] MSE: 0.012777967733875177
alpha: 0.3 Coefficients: [ 0.03040786 -0.01865127 -0.00095397 -0.01931883 -0.02083772  0.01063341] MSE: 0.01277759861491287
alpha: 0.4 Coefficients: [ 0.03040765 -0.01865112 -0.00095396 -0.01931854 -0.02083735  0.01063322] MSE: 0.01277722952332949
alpha: 0.5 Coefficients: [ 0.03040744 -0.01865098 -0.00095395 -0.01931824 -0.02083698  0.01063302] MSE: 0.01277686045912283
alpha: 0.6 Coefficients: [ 0.03040723 -0.01865083 -0.00095394 -0.01931795 -0.0208366   0.01063283] MSE: 0.012776491422290617
alpha: 0.7 Coefficients: [ 0.03040701 -0.01865068 -0.00095393 -0.01931766 -0.02083623  0.01063263] MSE: 0.012776122412830678
alpha: 0.8 Coefficients: [ 0.0304068  -0.01865053 -0.00095392 -0.01931736 -0.02083585  0.01063243] MSE: 0.012775753430740755
alpha: 0.9 Coefficients: [ 0.03040659 -0.01865039 -0.00095391 -0.01931707 -0.02083548  0.01063224] MSE: 0.012775384476018612
alpha: 1.0 Coefficients: [ 0.03040638 -0.01865024 -0.0009539  -0.01931677 -0.02083511  0.01063204] MSE: 0.012775015548662021
```

Here in ridge regression, we can see that as the value of alpha increases, MSE also increases due to the regularization term alpha. We performed PCA and plotted the coefficients of the top 6 principal components. For each principal coefficient, we can see that the coefficient value is decreasing with an increasing alpha value. That means we can say that it is slowly tending to zero. A decrease in the value of the coefficients of principal candidates indicates the effect of regularization.

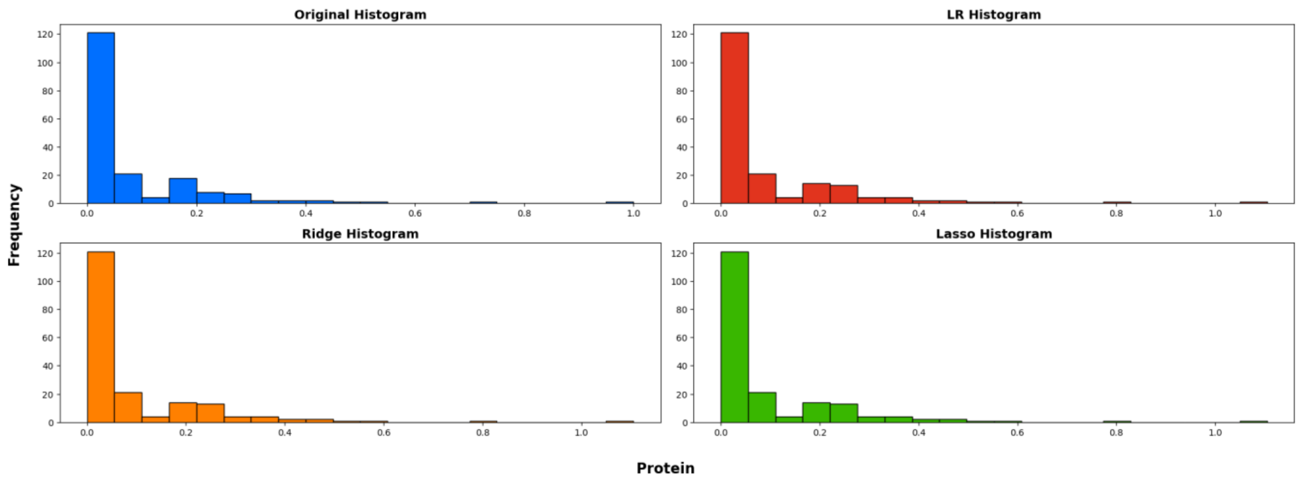
Lasso Regression

```python
def validation(actual, predicted):
    mse = np.mean((actual - predicted) ** 2)
    rmse = np.sqrt(mse)
    r2 = 1 - (np.sum((actual - predicted) ** 2) / np.sum((actual - np.mean(actual)) ** 2))

    return mse, rmse, r2, predicted

target = "Protein (G)"

lasso_1_df = pd.DataFrame(columns = ["alpha", "MSE", "RMSE", "R2"])
X = sl_df.drop([target] + string_columns, axis=1)
y = sl_df[target]

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

pca = PCA(n_components = 6)
X_pca = pca.fit_transform(X_scaled)

for j in np.arange(0.1, 1.1, 0.1):
    lasso_model = Lasso(alpha=round(j, 1))
    lasso_model.fit(X_pca, y)
    lasso_predicted = lasso_model.predict(pca.transform(scaler.transform(l_df.drop([target] + string_columns, axis=1))))

    mse, rmse, r2, predicted = validation(l_df[target], lasso_predicted)
    lasso_1_df = lasso_1_df.append({"alpha": round(j, 1), "MSE": mse, "RMSE": rmse, "R2": r2}, ignore_index=True)
    print("alpha:", round(j, 1), "Coefficients:", lasso_model.coef_, "MSE:", mse)

lasso_1_df = lasso_1_df.sort_values(by=["R2", "MSE"], ascending=[False, True])
```



```
alpha: 0.1 Coefficients: [ 0.01913052 -0.00587453 -0.         -0.         -0.          0.        ] MSE: 0.014734998891426241
alpha: 0.2 Coefficients: [ 0.00785254 -0.          -0.         -0.         -0.          0.        ] MSE: 0.017220800047892172
alpha: 0.3 Coefficients: [ 0. -0. -0. -0. -0.  0.] MSE: 0.02183712022282581
alpha: 0.4 Coefficients: [ 0. -0. -0. -0. -0.  0.] MSE: 0.02183712022282581
alpha: 0.5 Coefficients: [ 0. -0. -0. -0. -0.  0.] MSE: 0.02183712022282581
alpha: 0.6 Coefficients: [ 0. -0. -0. -0. -0.  0.] MSE: 0.02183712022282581
alpha: 0.7 Coefficients: [ 0. -0. -0. -0. -0.  0.] MSE: 0.02183712022282581
alpha: 0.8 Coefficients: [ 0. -0. -0. -0. -0.  0.] MSE: 0.02183712022282581
alpha: 0.9 Coefficients: [ 0. -0. -0. -0. -0.  0.] MSE: 0.02183712022282581
alpha: 1.0 Coefficients: [ 0. -0. -0. -0. -0.  0.] MSE: 0.02183712022282581
```

Here in Lasso regression, we can see that as the value of alpha increases, MSE also increases due to the regularization term alpha. We performed PCA and plotted the coefficients of the top 6 principal components. For each principal component, we can see that the value of the coefficient at staring is reduced to 0, and for the available principal coefficients, it decreases with an increasing alpha value. That means that the coefficients are moving fast to zero. A start with 0 and a decrease in the value of the coefficients of principal candidates indicates the effect of strong regularization in Lasso regression.



The 3 major parts of logistic regression works mostly the samein closely predicting the value of the original test data. We cannot see a major deviation in skewness in the data.