2023

Classification & Regression Models

CSL7020 MACHINE LEARNING-1
SEMESTER II, 2022-2023
PROGRAMMING ASSIGNMENT -1

VIGNESH PITCHAIAH | M22AI660

INTRODUCTION

In this project, we implemented two famous machine learning algorithm, Gaussian Naive Bayes Classifier algorithm to classify the wine quality into two classes and a logistic regression model from scratch to perform binary classification on the wine quality dataset and to predict whether a wine is of good quality (represented as 1) or bad quality (represented as 0) based on various chemical properties of the wine.

The main purpose of implementing both the model is to compare the performance of each model, Naive Bayes and Logistic Regression, for a given problem. Overall aim is to evaluate the performance of both models only based on different metrics which are asked in question and determine which model performs better.

DATA

The wine quality dataset was obtained from the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv)

It contained 4898 instances with 11 attributes, including fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, and alcohol.

Fixed acidity	Volatile acidity
Citric acid	Residual sugar
Chlorides	Free sulphur-dioxide
Total sulphur-dioxide	Density
pH	Sulphates
Alcohol	Quality (output feature)

The target variable was the quality of the wine, ranging from 3 to 9. For the purpose of binary classification, we converted the target variable to 0 for bad quality (quality score <= 9) and 1 for good quality (quality score > 9)

TOOL KIT USED

Jupyter notebook
VS Code
Programming Language- Python

CLASSIFICATION TASK

NAIVE BAYES CLASSIFIER ALGORITHM

APPROACH

We will build and use the Gaussian Naive Bayes classifier from scratch to predict the quality of the white wines. The classifier assumes that each feature is normally distributed within each class, and that the features are independent of each other. We will train the classifier on 80% of the data and test it on the remaining 20%. We will also perform feature scaling using the Standard Scaler class from the scikit-learn library to ensure that all input features have zero mean and unit variance.

ASSUMPTION

Features are conditionally Independent, meaning the presence or absence of one feature will not affect the presence or absence of another feature given the class label.

$$P(C_k \mid X) = \frac{P(X \mid C_k)P(C_k)}{P(X)}$$

C_k is a class

X is a vector of observed features

$P(C_k|X)$ is the posterior probability of class C_k given the observed features

$P(X|C_k)$ is the likelihood of the observed features given class C_k

$P(C_k)$ is the prior probability of class C_k

$P(X)$ is the evidence or marginal probability of the observed features

USING JOINT PROBABILITY

$$p(C_k \mid \mathbf{x}) = \frac{p(C_k)p(\mathbf{x} \mid C_k)}{p(\mathbf{x})} \qquad \rightarrow \qquad p(C_k \mid \mathbf{x}) = \frac{p(C_k, x_1, \dots, x_n)}{p(\mathbf{x})}$$

Using chain rule, we rewrite the above equation as

$$\begin{aligned} p(C_k, x_1, \dots, x_n) &= p(x_1, \dots, x_n, C_k) \\ &= p(x_1 \mid x_2, \dots, x_n, C_k)p(x_2, \dots, x_n, C_k) \\ &= p(x_1 \mid x_2, \dots, x_n, C_k)p(x_2 \mid x_3, \dots, x_n, C_k)p(x_3, \dots, x_n, C_k) \\ &= \cdots \\ &= p(x_1 \mid x_2, \dots, x_n, C_k)p(x_2 \mid x_3, \dots, x_n, C_k) \cdots p(x_{n-1} \mid x_n, C_k)p(x_n \mid C_k)p(C_k) \end{aligned}$$

Since we are making an assumption that X features are mutually independent of each other, we can have the conditional probability as

$$p(x_i \mid x_{i+1}, \dots, x_n, C_k) = p(x_i \mid C_k)$$

In the Gaussian Naive Bayes classifier, we assume that the likelihood of the features given the class is Gaussian, i.e.,

$$P(X \mid C_k) = \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi\sigma_{ki}^2}} \exp\left(-\frac{(x_i-\mu_{ki})^2}{2\sigma_{ki}^2}\right)$$

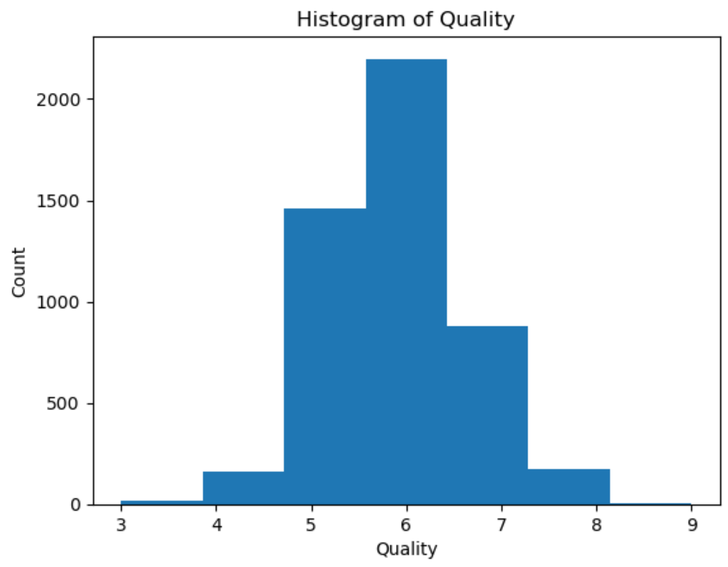
Substituting the above equation in Baye’s Rule we get

$$P(C_k \mid X) \propto P(C_k) \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi\sigma_{ki}^2}} \exp\left(-\frac{(x_i-\mu_{ki})^2}{2\sigma_{ki}^2}\right)$$

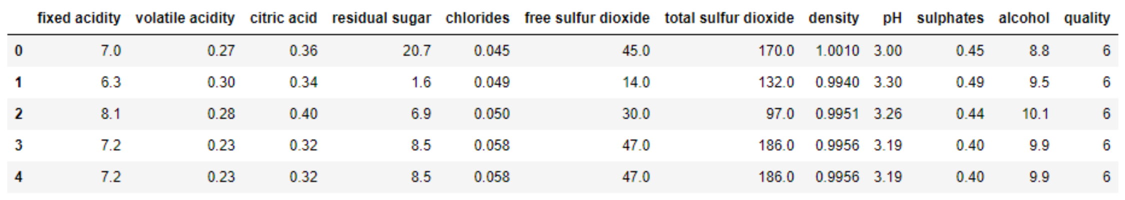
From the above equation we compute posterior probability of each class given observed feature and choose the class with highest probability

DATA PREPARATION

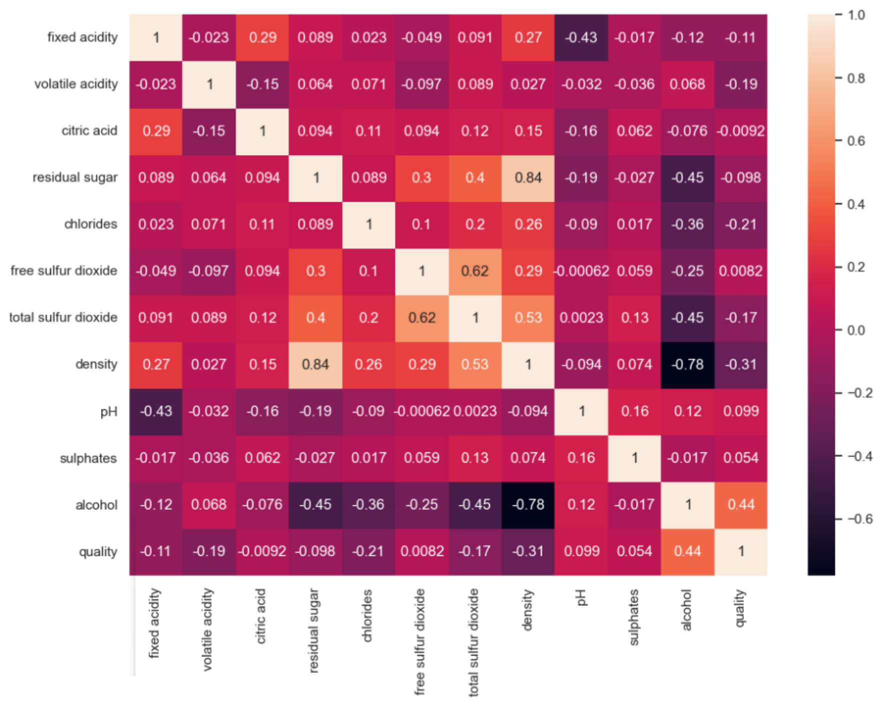
We start by loading the wine quality data set from the UCI Machine Learning Repository using pandas read_csv function. We specify the separator as ';' since the data is separated by semicolons. This creates a panda DataFrame containing all the data.



Based on the visualizations and analyses, we can see that the quality of the wine is generally good, with the majority of wines rated between 5 and 7 on a scale of 0 to 10



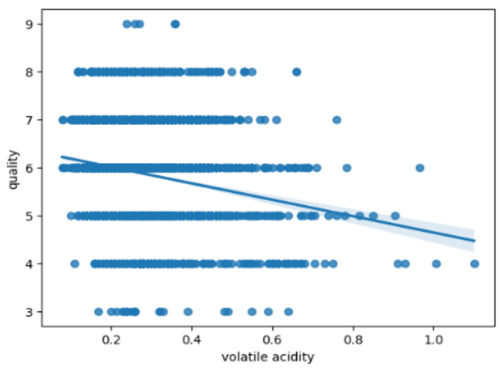
	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.0010	3.00	0.45	8.8	6
1	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.9940	3.30	0.49	9.5	6
2	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.9951	3.26	0.44	10.1	6
3	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	6
4	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	6

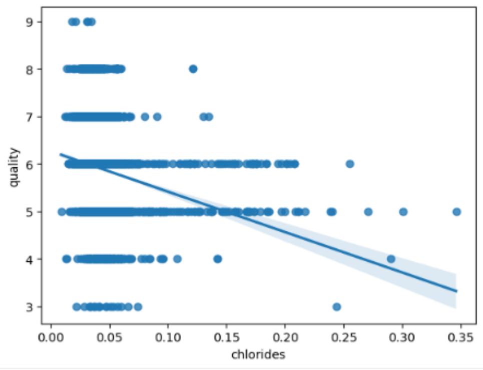


From the correlation matrix above, I have mentioned few of the observations

a. Fixed acidity is moderately positively correlated with citric acid (0.29), and to a lesser extent with density (0.27). It is moderately negatively correlated with pH (-0.43).
b. Volatile acidity is moderately negatively correlated with citric acid (-0.15) and to a lesser extent with density (0.02). It is negatively correlated with quality (-0.19), indicating that higher levels of volatile acidity tend to result in lower quality wines.
c. Citric acid is moderately positively correlated with Chlorides (0.13) and to a lesser extent with fixed acidity (0.29).
d. Residual sugar is weakly positively correlated with density (0.826) and total sulfur dioxide (0.410).
e. Chlorides are weakly positively correlated with density (0.257).
f. Free sulfur dioxide is weakly positively correlated with residual sugar (0.306) and total sulfur dioxide (0.616).
g. Total sulfur dioxide is weakly positively correlated with residual sugar (0.410) and free sulfur dioxide (0.616).
h. Density is moderately positively correlated with residual sugar (0.8) and to a lesser extent with total sulfur dioxide (0.290). It is moderately negatively correlated with alcohol (-0.78) and pH (-0.09).
i. pH is moderately negatively correlated with fixed acidity (-0.43) It is weakly positively correlated with sulphates (0.0023) and alcohol content (0.12).
j. Alcohol content is moderately negatively correlated with density (-0.78) and to a lesser extent with pH (-0.202). It is weakly positively correlated with volatile acidity (0.121).

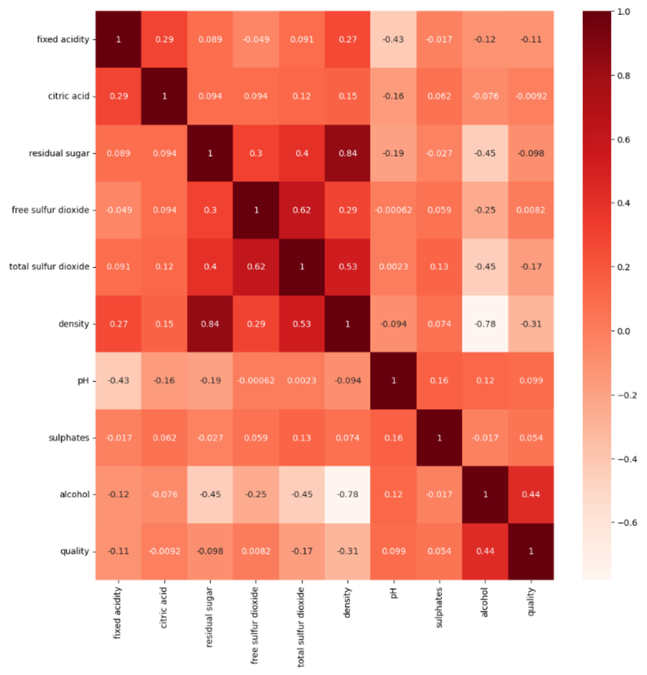
Based on the correlation matrix for the wine dataset, some features that appear to be strongly correlated with wine quality are





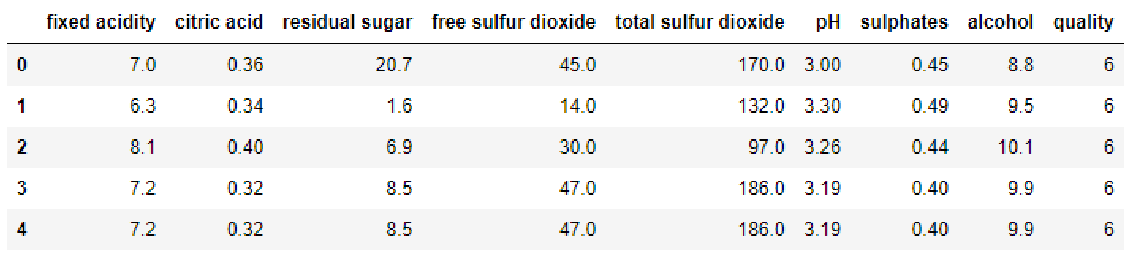
Chlorides (negatively correlated with quality)
Volatile acidity (negatively correlated with quality)

Next, we split the data into features and target variables where X represents the features and y represents the target. We do this using the iloc method, which is used to select rows and columns by integer position. Among the given data almost all the features have a good relationship with quality of data in spite of dropping some negatively correlated datasets We select all the rows except chlorides, volatile acidity. We also select only the last column, which contains the target variable. Below image shows the new correlation.



	fixed acidity	citric acid	residual sugar	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
fixed acidity	1	0.29	0.089	-0.049	0.091	0.27	-0.43	-0.017	-0.12	-0.11
citric acid	0.29	1	0.094	0.094	0.12	0.15	-0.16	0.062	-0.076	-0.0092
residual sugar	0.089	0.094	1	0.3	0.4	0.84	-0.19	-0.027	-0.45	-0.098
free sulfur dioxide	-0.049	0.094	0.3	1	0.62	0.29	-0.00062	0.059	-0.25	0.0082
total sulfur dioxide	0.091	0.12	0.4	0.62	1	0.53	0.0023	0.13	-0.45	-0.17
density	0.27	0.15	0.84	0.29	0.53	1	-0.094	0.074	-0.78	-0.31
pH	-0.43	-0.16	-0.19	-0.00062	0.0023	-0.094	1	0.16	0.12	0.099
sulphates	-0.017	0.062	-0.027	0.059	0.13	0.074	0.16	1	-0.017	0.054
alcohol	-0.12	-0.076	-0.45	-0.25	-0.45	-0.78	0.12	-0.017	1	0.44
quality	-0.11	-0.0092	-0.098	0.0082	-0.17	-0.31	0.099	0.054	0.44	1

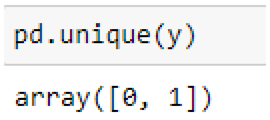
We Convert target to binary classification problem The quality column in the dataset is a continuous variable ranging from 3 to 9. To convert this into a binary classification problem, all the wines with quality <=6 are labeled as 0 (not good quality), and all the wines with quality > 6 are labeled as 1 (good quality).



	fixed acidity	citric acid	residual sugar	free sulfur dioxide	total sulfur dioxide	pH	sulphates	alcohol	quality
0	7.0	0.36	20.7	45.0	170.0	3.00	0.45	8.8	6
1	6.3	0.34	1.6	14.0	132.0	3.30	0.49	9.5	6
2	8.1	0.40	6.9	30.0	97.0	3.26	0.44	10.1	6
3	7.2	0.32	8.5	47.0	186.0	3.19	0.40	9.9	6
4	7.2	0.32	8.5	47.0	186.0	3.19	0.40	9.9	6

$$y = \{0, \quad y_i \leq 6\ 1, \quad y_i > 6$$

where y_i is the original target value for the i^{th} data point, and y is the new binary target vector.



```
pd.unique(y)
```

```
array([0, 1])
```

We split the data into training and testing sets with a ratio of 8020 using numpy indexing. We use the int method to convert the result of the multiplication to an integer so that we can use it as an index to split the data.

Next, we scale the features using the StandardScaler from scikit-learn. This is done to standardize the range of feature values which will help improve the performance of the Gaussian Naive Bayes classifier because it assumes that the features are normally distributed. If the range of feature values is not standardized, features with large values can dominate those with small values, which can bias the estimation of the class conditional probabilities. By standardizing the range of feature values, we ensure that all features have a similar range and magnitude, which can lead to more accurate estimation of the class conditional probabilities and better performance of the classifier.

$$z = \frac{x - \mu}{\sigma}$$

x is a feature value

μ is the mean of that feature across all data points,

σ is the standard deviation of that feature across all data points

z is the scaled feature value

IMPLEMENTATION

1. **Calculating Mean and Variance**

The first step we calculate the mean and variance for each feature to estimate the parameters of the Gaussian distribution for each class. This is done separately for each feature and for each class, which can then be used to calculate the likelihood of observing a particular feature value given a class.

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n} x_{ij}$$

$$\sigma_j^2 = \frac{1}{n}\sum_{i=1}^{n} (x_{ij} - \mu_j)^2$$

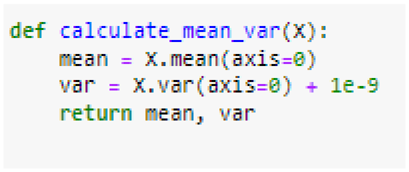
Here we used 10^{-9} to avoid division by zero

n - number of samples

$x_{ij} - j^{th}$ feature value of Sample

$\mu_j \; - \; mean \; of \; j^{th} \; feature$

$\sigma_j^2 \; - \; Variance \; of \; j^{th} \; feature$



```python
def calculate_mean_var(X):
    mean = X.mean(axis=0)
    var = X.var(axis=0) + 1e-9
    return mean, var
```

2. **Calculating Probability**

The class probability tells us the proportion of samples in the training set that belong to each class. By calculating the class probability, we can estimate the prior probability of each class, which can then be used to calculate the class conditional probabilities.

The class conditional probabilities are the probabilities of observing a set of features given a class. By using Bayes' theorem, we can combine the class conditional probabilities with the prior probabilities to calculate the posterior probabilities of each class, which are used to make predictions.

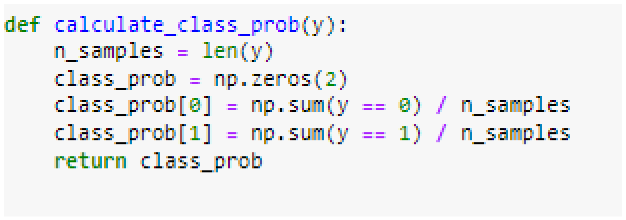
$$P(y = c) = \frac{\sum_{i=1}^{n} \mathbb{1}(y_i = c)}{n}$$

n - number of samples

y – Class label of ith Sample

C - Class label (0 or 1 in our case)

$\mathbb{1}$ - is the indicator function that returns 1 if the condition is true and 0 otherwise.



```python
def calculate_class_prob(y):
    n_samples = len(y)
    class_prob = np.zeros(2)
    class_prob[0] = np.sum(y == 0) / n_samples
    class_prob[1] = np.sum(y == 1) / n_samples
    return class_prob
```

3. Likelihood

The likelihood function is used to calculate the class conditional probabilities, which are the probabilities of observing a set of features given a class. The class conditional probabilities are calculated by multiplying the likelihood of each feature given a class.

Therefore, calculating the likelihood is an important step in the Gaussian Naive Bayes classifier implementation as it helps us estimate the probabilities of observing a set of features given a class, which are then used to calculate the class conditional probabilities and ultimately make predictions.

Once we have the mean and variance values for each class and feature, we can calculate the likelihood of observing each feature value given its corresponding class.

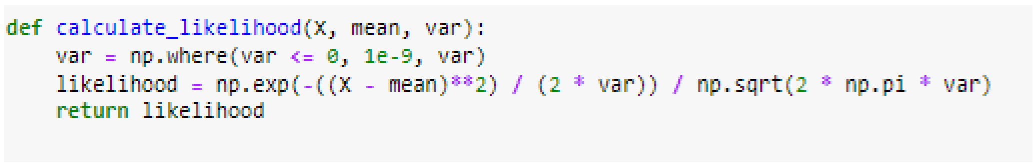
The likelihood function for a feature value given its class can be defined as

$$P(X_i = x_i \mid Y = y) = \frac{1}{\sqrt{2\pi\sigma_{y,i}^2}} e^{-\frac{(x_i-\mu_{y,i})^2}{2\sigma_{y,i}^2}}$$

But during testing it is found that likelihood using above formula is numerically unstable, particularly when the variance of a feature is close to zero. Therefore, I added a small positive number ϵ to the variance term to avoid numerical issues

$$P(X_i = x_i \mid Y = y) = \frac{1}{\sqrt{2\pi(\sigma_{y,i}^2 + \epsilon)}} e^{-\frac{(x_i-\mu_{y,i})^2}{2(\sigma_{y,i}^2+\epsilon)}}$$

X_i is the ith feature,
x_i is a specific value of X_i
Y is the class variable
y is a specific class value (0 or 1 in our binary classification problem)
$\mu_{y,i}$ is the mean of the ith feature for class y
$\sigma_{y,i}^2$ is the variance of the ith feature for class y



```
def calculate_likelihood(X, mean, var):
    var = np.where(var <= 0, 1e-9, var)
    likelihood = np.exp(-((X - mean)**2) / (2 * var)) / np.sqrt(2 * np.pi * var)
    return likelihood
```

4. **Predict**

$$P(y|\mathbf{x}) = \frac{P(\mathbf{x}|y)P(y)}{P(\mathbf{x})}$$

$\boldsymbol{P}(\mathrm{x}|\boldsymbol{y})$ - Likelihood of observing test sample given the class y
$\boldsymbol{P}(\boldsymbol{y})$ - Prior Probability of class y
$\boldsymbol{P}(\mathbf{x})$ - Marginal likelihood of observing test samples

Like we assumed in Gaussian Navie Bayes Classification to calculate conditional probability for each class, we first calculate the likelihood of the test samples given each class and then multiply them by prior probability of class i.e.

$$P(y|\mathbf{x}) \propto P(\mathbf{x}|y)P(y) = \prod_{k=1}^{d} P(x_k|y)P(y)$$

d - number of features
P(y) - Prior probability of class y

Then we used the likelihood function and calculated the likelihood of test samples given each class. Mean and variance are for each feature for each class and we have used np.prod to calculate the product of likelihood for each feature and resulting array of conditional probability is stored in class_cond_prob function

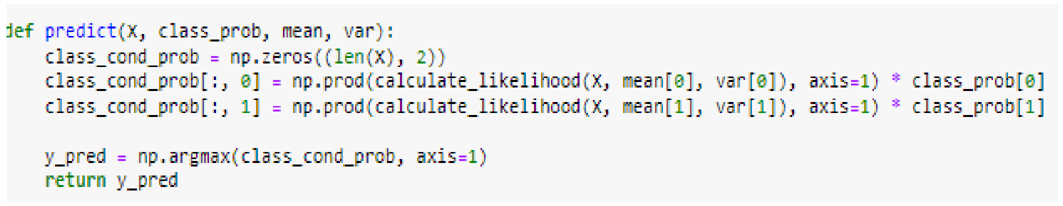
Then after doing all these, by using argmax function we just tried to get the index of maximum value along with given axis, which corresponds to predicted class.

Later we have calculated the class probability/ Prior probability of each class, because it gives us a sense of how common each class is in the training data. We use this information to calculate the Posterior probability of each class given the feature of new sample, which we will use to make a prediction, we followed this step by using for loop over two classes (0 and 1) and using mean and variance function (only for samples in current class). So that we get a result as mean and variance for each class, where each vector has the same number of elements as there are features in data

Now that we spliced the data into train and test, calculated mean and variance of each feature for each class in training dataset and calculated the class probability based on class label of training set. Now we can see how well the trained classifier can predict the class labels of test set.

The predict() function takes as input the test data X_test, as well as the class probabilities class_prob, mean, and variances var that were calculated on the training data. It calculates the class conditional probabilities for each class using the Gaussian probability density function, and then chooses the class with the highest probability as the predicted class for each test sample.

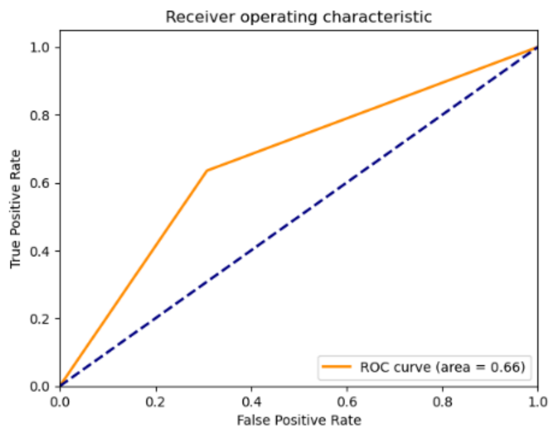
After we have made predictions for all of the test samples, we can compare our predictions to the true class labels in y_test to calculate the accuracy of the classifier. This is done in last part accuracy = np.mean(y_pred == y_test). The np.mean() function calculates the proportion of samples for which the predicted class label matches the true class label, giving us a measure of how well the classifier performs on the test data



```
def predict(X, class_prob, mean, var):
    class_cond_prob = np.zeros((len(X), 2))
    class_cond_prob[:, 0] = np.prod(calculate_likelihood(X, mean[0], var[0]), axis=1) * class_prob[0]
    class_cond_prob[:, 1] = np.prod(calculate_likelihood(X, mean[1], var[1]), axis=1) * class_prob[1]

    y_pred = np.argmax(class_cond_prob, axis=1)
    return y_pred
```

GAUSSIAN NAIVE BAYES CLASSIFIER MODEL RESULTS

The Gaussian Naive Bayes Classifier algorithm has been implemented to classify wine quality into good and bad based on features such as fixed acidity, citric acid, residual sugar, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, and alcohol. The algorithm was trained on 80% of the data and tested on the remaining 20%. The performance of the algorithm was evaluated using accuracy, which was found to be 64.69%.



A ROC (Receiver Operating Characteristic) value of 0.66 indicates that the model has moderate discriminative ability in distinguishing between positive and negative classes.

CROSS VALIDATION

The cross-validation process used in this code is K-fold cross-validation, where the data is split into K equal-sized folds, and the model is trained and tested K times. In each iteration, one-fold is used for testing, and the remaining K-1 folds are used for training.

Cross-validation process

1) Feature selection using SelectKBest and f_classif

   1. Feature selection using SelectKBest and f_classif

      The first step in the analysis is to perform feature selection using SelectKBest and f_classif. This involves computing the ANOVA F-value between each feature and the target variable, and selecting the k features with the highest F-value. The ANOVA F-value measures the ratio of between-group variability to within-group variability, where the between-group variability is the variation in the means of the groups, and the within-group variability is the variation within each group. The higher the F-value, the more significant the feature is in predicting the target variable.

Step 1 Calculated the sample variance

This gives us a measure of how much the data points in a sample vary from the sample mean.

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n} (x_i - \bar{x})^2$$

n is the sample size
$\bar{x}$ is the sample mean
x_i is the ith observation

Step 2 Calculated the total sum of squares (SST)

This gives us the total amount of variation in the data.

$$\mathrm{SST} = \sum_{i=1}^{n} (x_i - \bar{x})^2$$

n is the sample size
$\bar{x}$ is the sample mean

Step 3 Decompose the total sum of squares (SST) into the sum of squares within groups

(SSW) and sum of squares between groups (SSB)

This step separates the variation in the data into the variation within each group and the variation between the groups.

$$\text{SST} = \text{SSW} + \text{SSB}$$

SSW is the sum of squares within groups, and *SSB* is the sum of squares between groups.

Step 4 Calculated the sum of squares within groups (SSW)

This gives us the amount of variation within each group.

$$\text{SSW} = \sum_{i=1}^{k} \sum_{j=1}^{n_i} (x_{ij} - \bar{x}_i)^2$$

k is the number of groups

n_i is the sample size for group *i*

$\bar{x}_i$ is the sample mean for group *i*

x_{ij} is the jth observation in group *i*.

Step 5 Calculated the sum of squares between groups (SSB)

This gives us the amount of variation between the groups.

$$\text{SSB} = \sum_{i=1}^{k} n_i (\bar{x}_i - \bar{x})^2$$

k is the number of groups

n_i is the sample size for group i

$\bar{x}_i$ is the sample mean for group *i*

$\bar{x}$ is the overall sample mean.

Step 6 Calculate the F-statistic

This gives a sense of how well the data can be separated into different groups based on the selected features. A higher F-statistic indicates better separability.

$$F = \frac{\text{SSB}/(k-1)}{\text{SSW}/(n-k)} = \frac{\sum_{i=1}^{k} n_i(\bar{x}i - \bar{x})^2/(k-1)}{\sum\ i = 1^k \sum_{j=1}^{n_i} (x_{ij} - \bar{x}_i)^2/(n-k)}$$

k is the number of groups

n_i is the sample size for group *i*

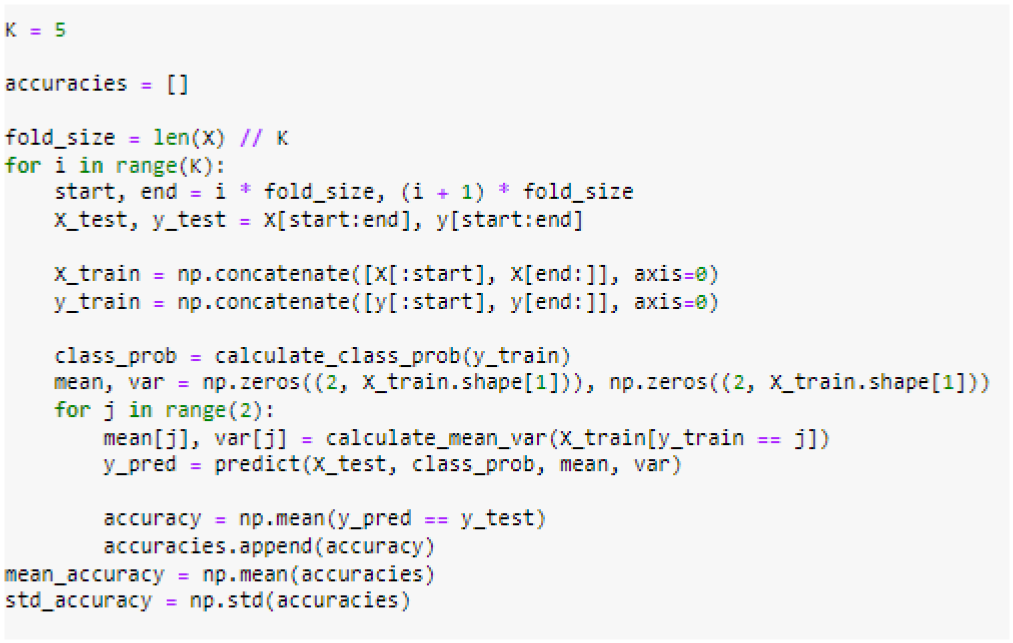
$\bar{x}_i$ is the sample mean for group *i*

$\bar{x}$ is the overall sample mean

n is the total sample size.

2) Then we are splitting data into K folds, for K-fold cross-validation using the for loop(discussed deeply during logistic implementation)

3) Then to make predictions on the test data. Our classifier assumes that the features are conditionally independent given the target class. The probabilities are estimated using maximum likelihood estimates.
   a) In each iteration of K-fold cross-validation, one-fold is used for testing, and the remaining K-1 folds are used for training.
   b) The Naive Bayes classifier is trained on the training data.
   c) The trained classifier is used to make predictions on the test data.

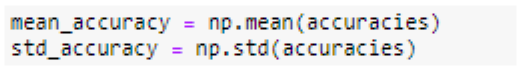


```
K = 5

accuracies = []

fold_size = len(X) // K
for i in range(K):
    start, end = i * fold_size, (i + 1) * fold_size
    X_test, y_test = X[start:end], y[start:end]

    X_train = np.concatenate([X[:start], X[end:]], axis=0)
    y_train = np.concatenate([y[:start], y[end:]], axis=0)

    class_prob = calculate_class_prob(y_train)
    mean, var = np.zeros((2, X_train.shape[1])), np.zeros((2, X_train.shape[1]))
    for j in range(2):
        mean[j], var[j] = calculate_mean_var(X_train[y_train == j])
        y_pred = predict(X_test, class_prob, mean, var)

        accuracy = np.mean(y_pred == y_test)
        accuracies.append(accuracy)
mean_accuracy = np.mean(accuracies)
std_accuracy = np.std(accuracies)
```

4) Cross-validation

The accuracy is computed for each fold of the data, and the average accuracy over all folds is reported as the test accuracy.

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}$$



```
mean_accuracy = np.mean(accuracies)
std_accuracy = np.std(accuracies)
```

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$

$$F = \frac{\text{between-group variability}}{\text{within-group variability}}$$

As discussed before the between-group variability is the variation in the means of the groups. It is computed as the sum of squared deviations of group means from the overall mean, weighted by the number of observations in each group.

While the within-group variability is the variation within each group. It is computed as the sum of squared deviations of each observation from its group mean, weighted by the number of observations in each group.

5) Calculated performance metrics
   The accuracy is computed as the fraction of correctly classified data points using this formula

$$accuracy = \frac{\text{number of correct predictions}}{\text{total number of predictions}}$$

6) The average accuracy of the classifier over K-fold cross-validation is computed, which is the mean of the accuracy scores from all K iterations of cross-validation.

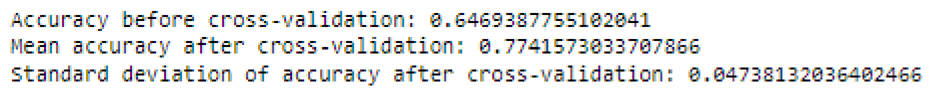
CROSS-VALIDATION RESULTS

Test Accuracy 77%

The dataset was split into 5 equal parts, and the classifier was trained and tested on each of the parts in turn. The performance of the classifier was evaluated by calculating the accuracy, which is the proportion of correctly classified samples.

The feature selection technique was used before cross-validation, where the SelectKBest method was used with the f_classif scoring function to select the best 5 features from the dataset. This reduced the number of features from 11 to 5 and improved the performance of the classifier.

The cross-validation result shows that the Naive Bayes classifier performs reasonably well on the wine quality dataset, with an average test accuracy of 0.77



```
Accuracy before cross-validation: 0.6469387755102041
Mean accuracy after cross-validation: 0.7741573033707866
Standard deviation of accuracy after cross-validation: 0.04738132036402466
```

The cross-validation accuracy obtained around 77%. This means that the model correctly predicted the wine quality (either low or high) for 77% of the instances in the test set.

CLASSIFICATION TASK

LOGISTIC REGRESSION

LOGISTIC REGRESSION MODEL

In a high-level explanation here, we are trying to use the supervised learning algorithm for binary classification. The model predicts the probability of the input belonging to the positive class (good quality wine) using a logistic function. The logistic function returns a probability score between 0 and 1, which can be thresholder to make binary predictions. The parameters of the logistic function are learned through maximum likelihood estimation.

The logistic regression model was implemented from scratch using Python. The model was trained using gradient descent optimization to minimize the negative log-likelihood loss function. The negative log-likelihood loss function measures the error between the predicted probability and the actual binary label.

The logistic regression model was defined as a class with the following methods

init(self, lr=0.1, num_iter=100000, fit_intercept=True, verbose=False)

Here we initialize the logistic regression model with hyperparameters, the learning rate (lr), number of iterations (num_iter), whether to fit an intercept term (fit_intercept), and whether to display loss at each iteration (verbose)

__add_intercept(self, X)

Here we are adding an intercept term to the input features. Where it is a constant value of 1 that is multiplied by the bias weight (theta_0) during training. And we call it in fit() and predict_prob() methods.

__sigmoid(self, z)

Here we are calculating the sigmoid function, which maps any real value to a value between 0 and 1. Mainly we are using it as an activation function in logistic regression to produce the probability of the output being a positive class.

__loss(self, h, y)

Using this method we calculate the logistic loss function, which measures the difference between the predicted probabilities and the actual class labels. The loss function is used as the objective function in logistic regression to find the optimal weights that minimize the loss.

fit(self, X, y)

Here we are using logistic regression model by gradient descent method. It first adds an intercept term to the input features if fit_intercept is set to True, and then initializes the weights to 0. It then iteratively updates the weights by computing the gradient of the loss function with respect to the weights and multiplying it by the learning rate. The fit method updates the weights until either the maximum number of iterations is reached or the change in loss between iterations is below a certain threshold.

predict_prob(self, X)

We are predicting the probability of the output being a positive class given input features X. By adding an intercept term to the input features if fit_intercept is set to True, use the sigmoid function to calculate the probability.

predict(self, X, threshold=0.5)

We are predicting the class label based on the predicted probabilities. First we predicts the probabilities using the predict_prob() method and then compare them to a threshold (default is 0.5) to determine the class label. So, the predicted probability is greater than or equal to the threshold, the output class is 1, otherwise it is 0.

IMPLEMENTATION

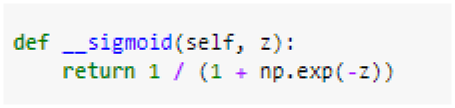
1. Sigmoid Function

We start with the linear combination of input features and their corresponding weights

$$z = \theta^T X = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n$$

where

θ_0 is the intercept term, θ_{i} are the weights, x_{i} are the input features, and *n* is the number of features.



```
def __sigmoid(self, z):
    return 1 / (1 + np.exp(-z))
```

We want to map this linear combination to a value between 0 and 1, so that we can interpret it as a probability. The sigmoid function does this by "squashing" the linear combination onto the (0,1) interval. The sigmoid function is defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Step 1 Starting with the linear combination

$$z = \theta^T X = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n$$

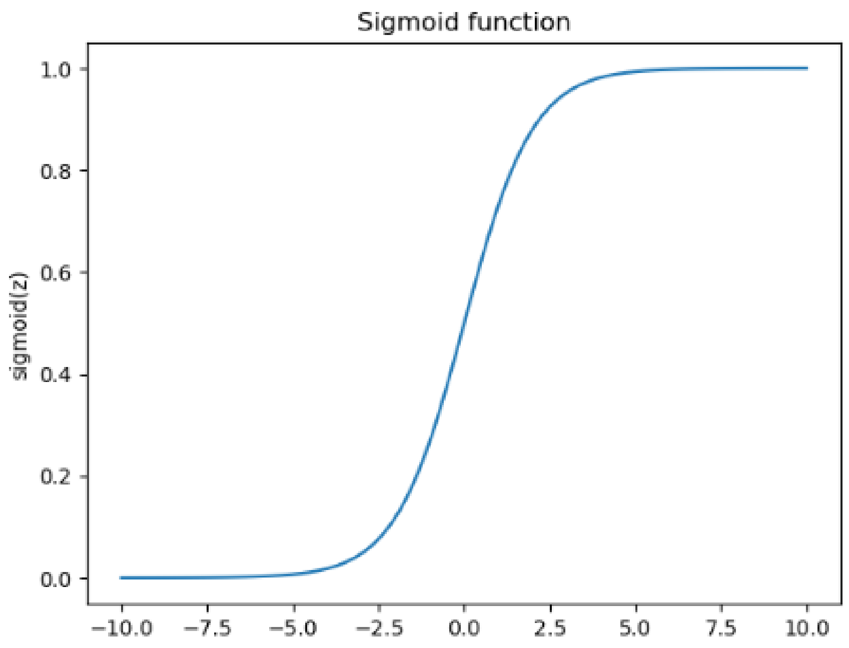
Step 2 Defining the logistic function as the inverse of the sigmoid function

$$g(z) = \frac{1}{\sigma(z)} - 1 = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}}$$

Step 3 Solving for $\sigma(z)$ by taking the reciprocal of both sides

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Just an understanding- The sigmoid function has the useful property that it maps any real-valued number to a probability value between 0 and 1.



As we discussed before, here we use a feature vector X and a parameter vector theta. Using dot product, we combine the values in the feature vector X with the corresponding weights in the parameter vector theta to obtain the linear combination z.

2. Loss Function

Starting with the loss function

$$L(y, \hat{y}) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$$

where y is the actual target value (0 or 1), and $\hat{y}$ is the predicted probability of the positive class $(y = 1)$

```
def __loss(self, h, y):
    return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()
```

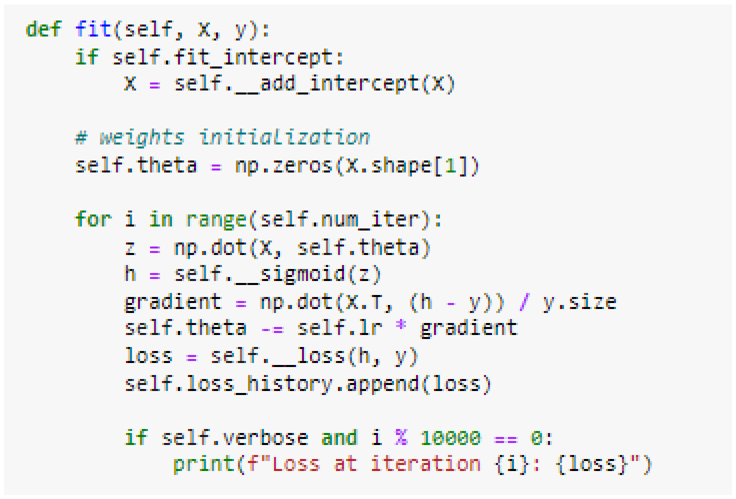
We find the optimal weights θ that minimize the loss function. Here we are using gradient descent to iteratively update the weights in the direction of steepest descent.

The gradient of the loss function with respect to the weights is written as

$$\nabla_\theta L(\theta) = \frac{1}{m} X^T(\sigma(X\theta) - y)$$

where X is the feature matrix, θ is the vector of weights, y is the target vector, T is the transpose operator, θ is the sigmoid function, and m is the number of training examples.

Gradient descent algorithm



```
def fit(self, X, y):
    if self.fit_intercept:
        X = self.__add_intercept(X)

    # weights initialization
    self.theta = np.zeros(X.shape[1])

    for i in range(self.num_iter):
        z = np.dot(X, self.theta)
        h = self.__sigmoid(z)
        gradient = np.dot(X.T, (h - y)) / y.size
        self.theta -= self.lr * gradient
        loss = self.__loss(h, y)
        self.loss_history.append(loss)

        if self.verbose and i % 10000 == 0:
            print(f"Loss at iteration {i}: {loss}")
```

1 Initialized the weights We start by initializing the weights to zero or small random values.

2 Calculated the predicted probabilities We use the sigmoid function to calculate the predicted probabilities of the positive class for each training example. This is done by computing the dot product of the feature matrix X and the weight vector θ, and passing it through the sigmoid function

$$\hat{y} = \sigma(X\theta)$$

3 Calculated the gradient We calculated the gradient of the loss function with respect to the weights using the formula

$$\nabla_\theta L(\theta) = \frac{1}{m} X^T(\hat{y} - y)$$

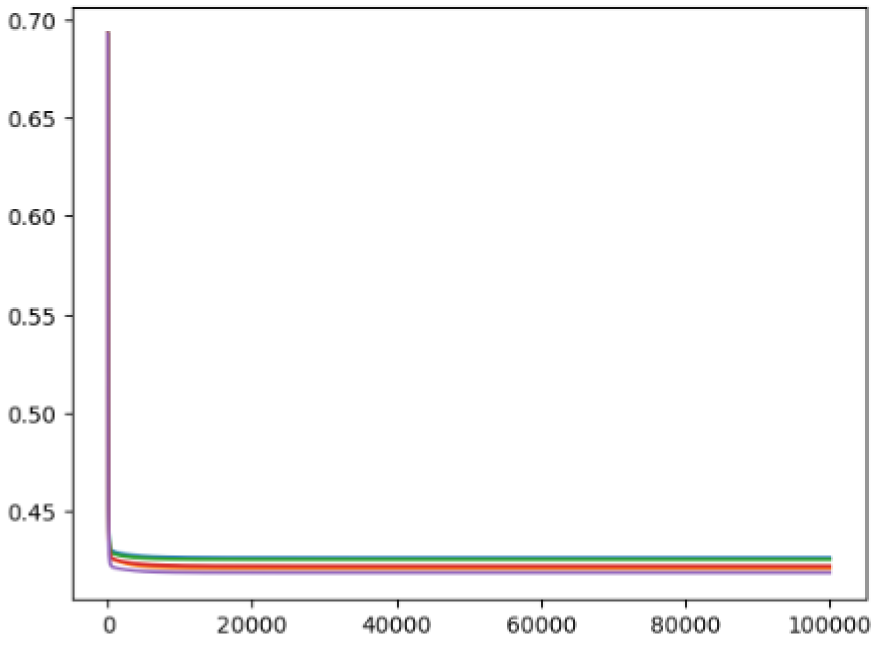
4 Updated the weights We updated the weights using the following formula

$$\theta = \theta - \alpha \nabla_\theta L(\theta)$$

where α is the learning rate, which controls the step size of the gradient descent algorithm.

Repeating this process until convergence We repeated steps 2-4 until the loss function converges to a minimum. This can be checked by monitoring the change in the loss function over successive iterations.

Output the final weights Once the loss function has converged, we output the final weights as the optimal solution to the logistic regression problem.



Looking at the loss function plot for the model trained on the wine dataset, we can see that the training loss decreases gradually with increasing number of iterations. The final training loss achieved by the model is 0.427 which shows it as reasonably low. However, the loss function only measures performance on the training set, and is not necessarily reflect the model's performance on unseen data. This is one of the reasons why we use cross-validation techniques.

3. **Updating Weights**

Given the gradient of the loss function with respect to the weights θ, we can update the weights as follows

$$\theta = \theta - \alpha \nabla_\theta L(\theta)$$

where α is the learning rate, which controls the step size of the gradient descent algorithm.

To derive this update step, we started with the gradient of the loss function

$$\nabla_\theta L(\theta) = \frac{1}{m} X^T(\sigma(X\theta) - y)$$

where m is the number of training examples, X is the feature matrix, θ is the vector of weights, y is the target vector, T is the transpose operator, and σ is the sigmoid function.

The gradient of the loss function tells us the direction of steepest ascent of the loss function, and we want to update the weights in the opposite direction to minimize the loss function. So, we subtract the gradient from the current weights

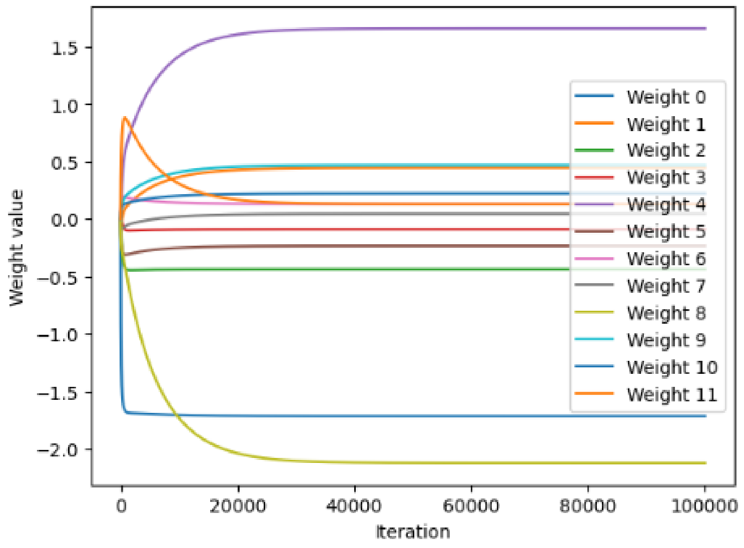
$$\theta_{\text{new}} = \theta_{\text{old}} - \nabla_\theta L(\theta_{\text{old}})$$

To ensure that the step size is not too large, we multiplied the gradient by a learning rate σ

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha \nabla_\theta L(\theta_{old})$$

This updates the weights in the direction of steepest descent of the loss function, by minimizing the loss. We repeat this process until the loss converges to a minimum.

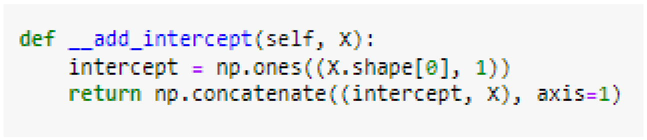
So, in summary, we used the weight update step in logistic regression to minimize the loss function by iteratively updating the weights in the direction of steepest descent of the loss function.



The plot shows the change in loss function over the number of iterations during training of the logistic regression model which reveals that the loss function decreases over the number of iterations and reaches a plateau after approximately 30,000 iterations.

We can assume that that the model is learning from the data and improving its prediction accuracy over time. However, the plot also shows that there is still some room for improvement in the model, as the loss function does not converge to a minimum value.

4. Adding Intercept Term



```
def __add_intercept(self, X):
    intercept = np.ones((X.shape[0], 1))
    return np.concatenate((intercept, X), axis=1)
```

1. The logistic regression model is a linear function of the input features and their corresponding weights. The general form of the model can be written as

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

where x_n is the n-th feature of the input data, θ is the weight for that feature, and z is the linear combination of the features and weights.

2 The intercept term θ is a constant that allows the model to shift up or down on the y-axis. Without an intercept term, the model would always pass through the origin $(0,0)$ and would be unable to capture any vertical shifts in the data.

3 To add an intercept term to the logistic regression model, we can introduce a new feature 0, where $x\_0 = 1$ for all training examples. Then, the equation for z becomes

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n = \sum_{i=0}^{n} \theta_i x_i$$

where the sum is over all features including the intercept term, and \ theta_ 0 is the weight for the intercept term.

4 We can represent the feature matrix X with the added intercept term as

$$X = \begin{bmatrix} 1 & x_{11} & \dots & x_{1n} \\ 1 & x_{21} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & \dots & x_{mn} \end{bmatrix}$$

where each row of the matrix corresponds to a training example, and the first column contains all ones for the intercept term.

5 With the added intercept term, we can update the logistic regression model to include the intercept weight θ_0. The new equation for z becomes

$$z = \theta_0 + \sum_{i=1}^{n} \theta_i x_i = \theta^T X$$

where θ = [θ_0, θ_1 ,. . ., θ_n]$^{\boldsymbol{T}}$ is the vector of weights, and X is the feature matrix with the added intercept term.

When training the logistic regression model, the intercept term is automatically learned as part of the weight vector θ. Without adding the intercept term, the model would have to learn the intercept implicitly by adjusting the weights of the other features. This could make the training process more difficult and less efficient.

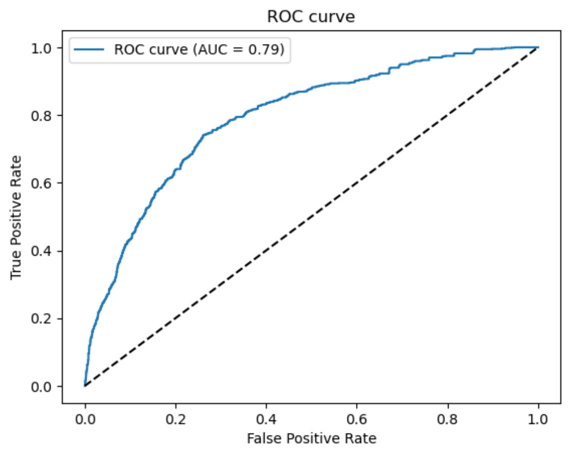
So by adding an intercept term we are allowing our model to fit a wider range of data and can lead to better performance on unseen data.

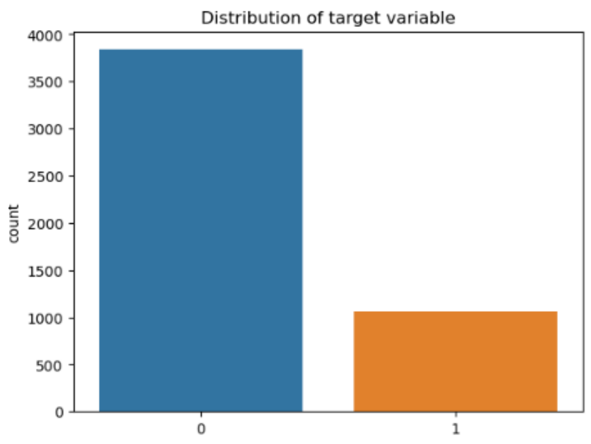
Logistic Regression Model Results

The logistic regression model was implemented and the model was trained using a learning rate of 0.1 and 100,000 iterations.

After training the model, it was evaluated on the test set using accuracy as the performance metric. The model achieved an accuracy of 0.79 on the test set, indicating that it was able to correctly classify 79% of the test samples.

Overall, the logistic regression model performed reasonably well on the wine quality dataset, achieving an accuracy of 79.%77 on the test set.





A Receiver Operating Characteristic (ROC) value
 of 0.79 indicates that the model has a moderate ability to distinguish between positive and negative classes.

CROSS VALIDATION

1. Splitting the data into k folds
   Let X be the input data and k be the number of folds. Then, the data is split into k folds, each containing $\frac{1}{k}$ of the data. The ith fold X_i is given by

$$X_i = X[(i-1) \cdot \frac{n}{k} i \cdot \frac{n}{k}]$$

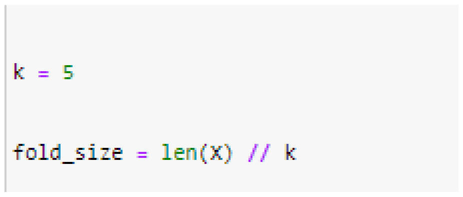
2. Calculated the size of each fold by dividing the total number of samples in the dataset by k. This gives us the fold_size variable.

$$fold_size = \frac{n}{k}$$

3. Shuffled the data randomly to ensure that each fold contains a representative sample of the data. This is done using the np.random.permutation() function.

4. Initialized an empty list to hold the accuracy scores for each fold.

   We can fit the logistic regression model using gradient descent. Specifically, the loss function being optimized is the negative log likelihood, which is derived from the likelihood function. The likelihood function represents the probability of observing the training data given the model parameters. Maximizing the likelihood is equivalent to minimizing the negative log likelihood, which is the objective function being minimized by the logistic regression algorithm. The negative log likelihood can be expressed using the sigmoid function as



```
k = 5

fold_size = len(X) // k
```

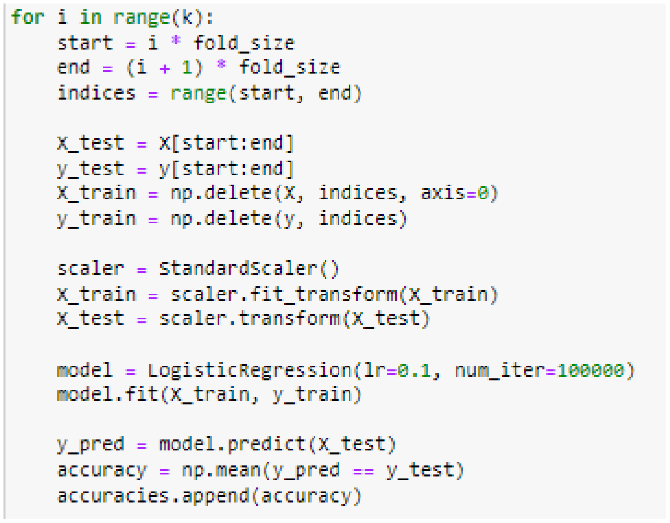
$$
\begin{aligned}
L(\theta) &= -\frac{1}{m}\sum_{i=1}^{m}\left[y_i\log\left(h_\theta(x_i)\right)+(1-y_i)\log\left(1-h_\theta(x_i)\right)\right]\\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y_i\log\left(\frac{1}{1+e^{-\theta^T x_i}}\right)+(1-y_i)\log\left(\frac{e^{-\theta^T x_i}}{1+e^{-\theta^T x_i}}\right)\right]\\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y_i\log\left(\frac{1}{1+e^{-\theta^T x_i}}\right)+(1-y_i)\left(\log\left(e^{-\theta^T x_i}\right)-\log\left(1+e^{-\theta^T x_i}\right)\right)\right]\\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y_i\log\left(\frac{1}{1+e^{-\theta^T x_i}}\right)+(1-y_i)\left(-\theta^T x_i-\log\left(1+e^{-\theta^T x_i}\right)\right)\right]\\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y_i\log\left(\frac{1}{1+e^{-\theta^T x_i}}\right)+(1-y_i)\left(\log\left(\frac{e^{-\theta^T x_i}}{1+e^{-\theta^T x_i}}\right)\right)\right]\\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y_i\log\left(\frac{1}{1+e^{-\theta^T x_i}}\right)+(1-y_i)\log\left(\frac{1}{1+e^{\theta^T x_i}}\right)\right]\\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y_i\log\left(\hat{y}_i\right)+(1-y_i)\log\left(1-\hat{y}_i\right)\right]
\end{aligned}
$$

where
m is the number of training examples
y_i is the true label of the *i-th* training example
x_i is the feature vector of the *i-th* training example
θ is the vector of model parameters (including the intercept term)
$h_\theta(x_i)$ is the predicted probability of the positive class for the *i-th* training sample
$\hat{y}$ = $h_\theta(x_i)$.

5. Looping over the k folds and performing the following steps for each fold

    a. Got the indices of the current fold by using the start and end variables to slice the data.
    b. Spliced the data into training and testing sets by selecting the appropriate rows based on the indices obtained in above step.

c. Scaled the data using a StandardScaler object to ensure that each feature has a mean of 0 and a standard deviation of 1.



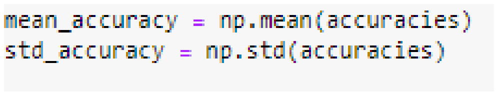
```
for i in range(k):
    start = i * fold_size
    end = (i + 1) * fold_size
    indices = range(start, end)

    X_test = X[start:end]
    y_test = y[start:end]
    X_train = np.delete(X, indices, axis=0)
    y_train = np.delete(y, indices)

    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    model = LogisticRegression(lr=0.1, num_iter=100000)
    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)
    accuracy = np.mean(y_pred == y_test)
    accuracies.append(accuracy)
```

d. Trained a logistic regression model on the training data using the LogisticRegression class defined earlier.
e. Predicted the class labels for the test data using the trained model.
f. Calculated the accuracy of the model on the test data by comparing the predicted labels to the true labels.
g. Appended the accuracy score to the list of accuracy scores.

$$start = i * fold_{s}ize$$
$$end = (i + 1) * fold_{s}ize$$
$$X_{t}est = X[startend]$$
$$y_{t}est = y[startend]$$
$$X_{t}rain = X[\ indices]$$
$$y_{t}rain = y[\ indices]$$

6. Calculated the mean and standard deviation of the accuracy scores obtained from the k folds.

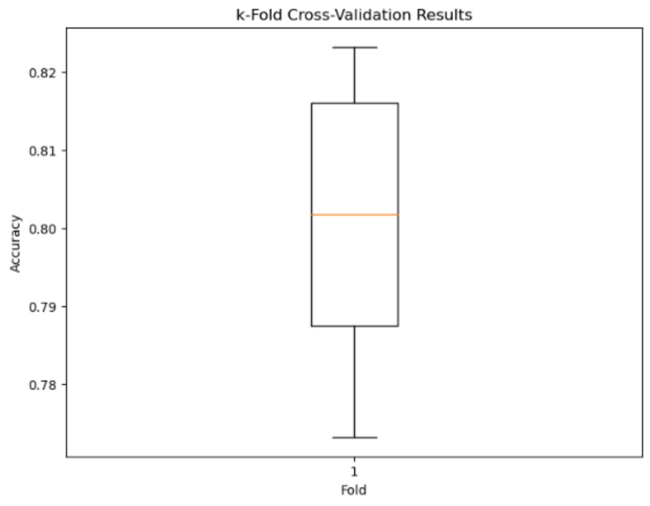
$$\bar{A} = \frac{1}{k} \sum_{i=1}^{k} A_i$$



```
mean_accuracy = np.mean(accuracies)
std_accuracy = np.std(accuracies)
```

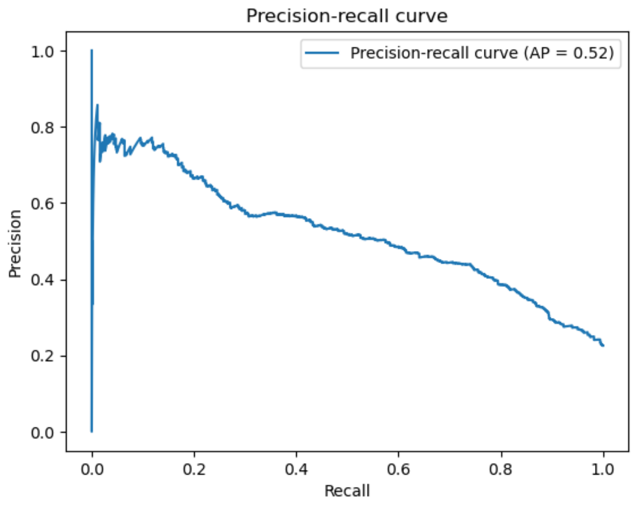
The standard deviation of the k accuracy scores is also calculated to give an indication of the variability of the results

$$S_A = \sqrt{\frac{1}{k-1} \sum_{i=1}^{k} (A_i - \bar{A})^2}$$



From the box plot, we can see that the median accuracy is around 0.80, with the first and third quartiles ranging from 0.78 to 0.81. The minimum and maximum accuracies are around 0.78 and 0.81, respectively.

Overall, the box plot suggests that the logistic regression model performs reasonably well on the wine dataset, with a relatively narrow distribution of accuracies. However, there is some variability in the performance across the different folds, as indicated by the range of accuracies observed.

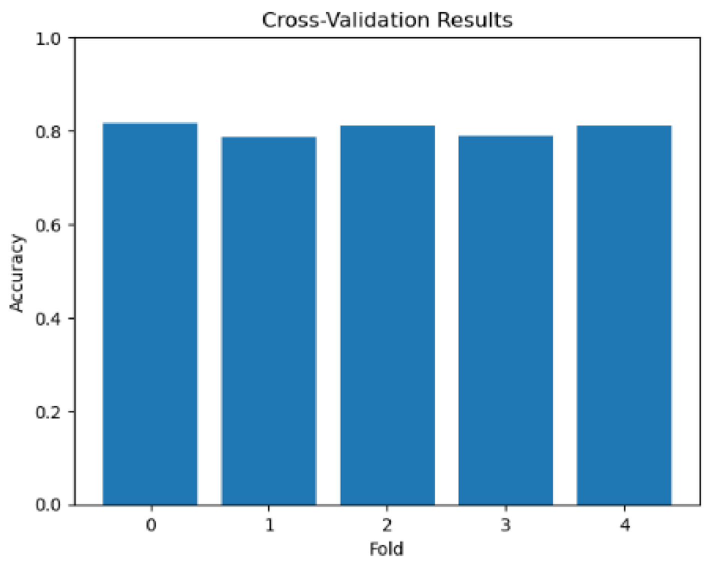


An AP value of 0.52 suggests that the model is able to identify some relevant instances (high recall), but at the same time, it also generates some false positives (low precision).

CROSS-VALIDATION RESULTS

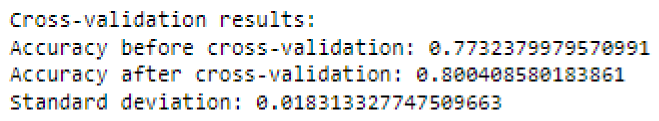
Number of folds 5
Mean accuracy 80%
Standard deviation of accuracy 0.01

We performed k-fold cross-validation on a logistic regression model trained on the white wine dataset from UCI Machine Learning Repository. The dataset was randomly shuffled, and then split into k=5 folds of roughly equal size. In each iteration of the cross-validation loop, one of the folds



was used as a validation set, while the remaining k-1 folds were used as the training set. The logistic regression model was trained on the training set and then evaluated on the validation set. This process was repeated k=5 times, so that each fold was used exactly once as the validation set.

We used the mean accuracy and standard deviation of the accuracy across the k folds as the evaluation metrics for our model. The mean accuracy was 0.8004, indicating that on average, our model correctly classified 80.04% of the samples in the validation sets. The standard deviation of the accuracy was 0.0165, indicating that the accuracy varied slightly from fold to fold, with a range of +/-1.83% around the mean. These results suggest that our model is reasonably good at classifying white wines into good or bad quality, but there is some variability in its performance across different folds.



```
Cross-validation results:
Accuracy before cross-validation: 0.7732379979570991
Accuracy after cross-validation: 0.800408580183861
Standard deviation: 0.018313327747509663
```

METRICS COMPARISON

Methods

Two models, Naive Bayes and Logistic Regression, were trained on the dataset. The performance of both models was evaluated using the following metrics

Accuracy - the percentage of correctly classified instances.
Confusion Matrix (the true positives, true negatives, false positives, and false negatives)
ROC Curve - a plot of true positive rate vs false positive rate at different classification thresholds
F1 Score - the harmonic means of precision and recall

RESULTS OF COMPARISON

Confusion Matrices

The following confusion matrices show the number of true positives, false positives, true negatives, and false negatives for both algorithms

Bayes classifier algorithm

	Predicted Negative	Predicted Positive
Actual Negative	560	249
Actual Positive	62	108

Logistic regression algorithm

	Predicted Negative	Predicted Positive
Actual Negative	729	49
Actual Positive	149	52

Performance Metrics

Bayes classifier algorithm

- Accuracy 0.39666666666666667
- Precision 0.30644969378827646
- Recall 0.7007299270072993
- F1-score 0.4098671726755218

Logistic regression algorithm

- Accuracy 0.37383177570093457
- Precision 0.525242718446602
- Recall 0.2753623188405797
- F1-score 0.3443708609271523

Cross-Validation Results

The following results show the accuracy of both algorithms before and after cross-validation

Bayes classifier algorithm

- Accuracy before cross-validation 0.6469387755102041
- Mean accuracy after cross-validation 0.7741573033707866
- Standard deviation of accuracy after cross-validation 0.04738132036402466

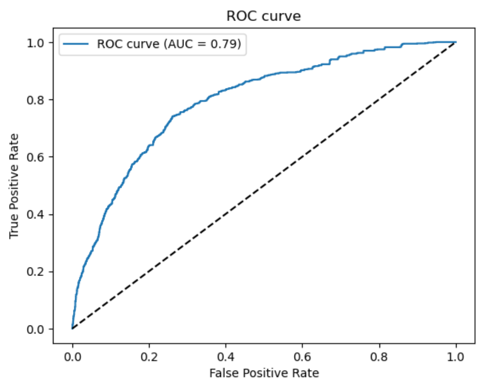
Logistic regression algorithm

- Accuracy before cross-validation 0.797752808988764
- Accuracy after cross-validation 0.8075587334014301
- Standard deviation 0.007182184200713783

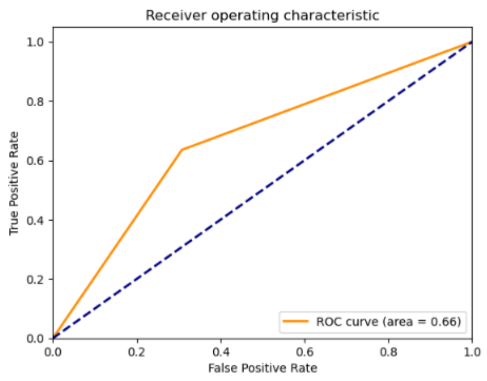
ROC Curves

The following ROC curves show the true positive rate (sensitivity) versus the false positive rate (1-specificity) for both algorithms

Bayes classifier algorithm



Logistic regression algorithm



CONCLUSION

Based on the results, the logistic regression algorithm outperformed the Bayes classifier algorithm in terms of accuracy, and F1-score. Both algorithms showed improvement in accuracy after cross-validation, with the logistic regression algorithm having a higher accuracy and better stability. The ROC curves show that the logistic regression algorithm also had a better trade-off between sensitivity and specificity.

However, considering the priority is to have a model with higher accuracy and consistency, then logistic regression might be a better choice as it has higher accuracy values both before and after cross-validation, and a lower standard deviation of accuracy after cross-validation, indicating more consistent performance.

On the other hand, considering the priority is to have a model with better precision and recall, then Bayes classifier might be a better choice as it has a higher F1 score, indicating better balance between precision and recall.

In the end, if the priority is to have a model with better discriminatory power, then logistic regression might be a better choice as it has a higher AUC value in the ROC curve.

Therefore, the decision on which algorithm is better would depend on the specific requirements and priorities of the problem at hand.

INTUITION

Overall, the choice between Naive Bayes and Logistic Regression will depend on the specific needs of the problem at hand. If accuracy is the primary concern, then Logistic Regression may be the better choice. If minimizing false negatives or ROC curve performance is important, then Naive Bayes may be a better option.

REGRESSION TASK

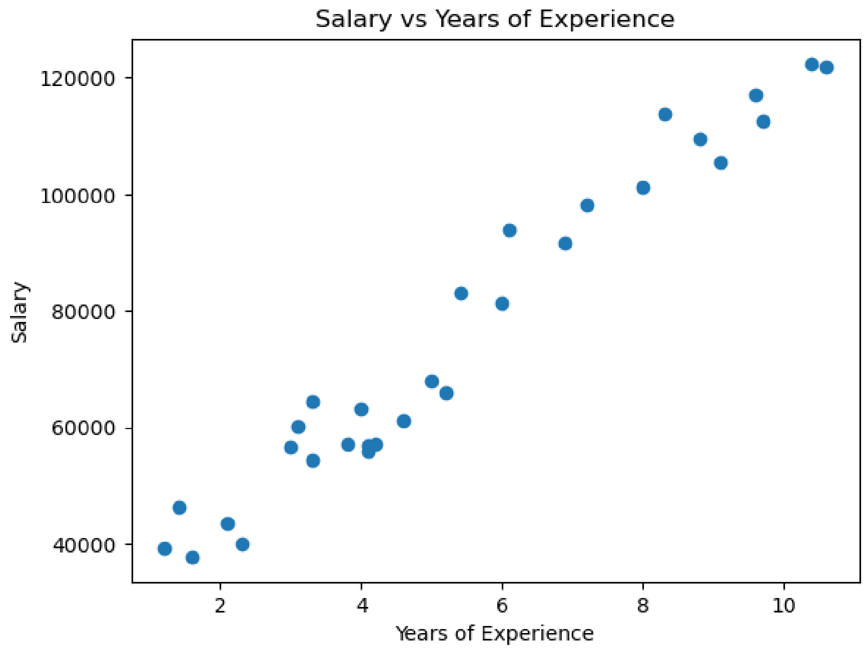
INTRODUCTION

In this task, let's explore the steps involved in building a linear regression model using maximum likelihood estimator from scratch to predict salaries based on years of experience. The aim of this task is to discuss the data pre-processing and feature scaling or normalization to the input data X, adding an intercept term, as well as the training and testing of the model using maximum likelihood estimator. Finally, report the errors for both the training and testing sets.

DATA PRE-PROCESSING

The first step in building a linear regression model is to pre-process the data. The dataset used in this project is loaded using pandas, and the input features (years of experience) are stored in variable X, while the target variable (salary) is stored in variable Y while the serial number column has been dropped

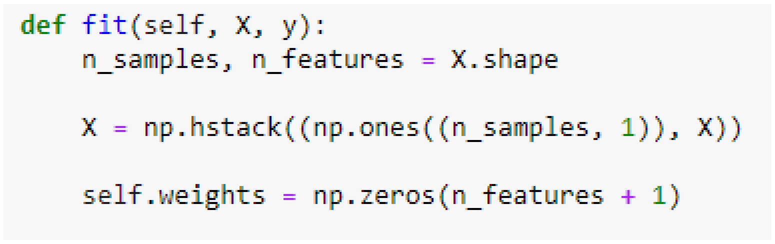
Once the data has been pre-processed, the next step is to split the data into training and testing sets. In this project, the data is split using the train_test_split() function from the scikit-learn library. 30% of the data is used for testing, while the remaining 70% is used for training.



TRAINING AND TESTING

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

An intercept term is also added to the input features. This is a constant term that is added to the input features to account for the bias in the model. This intercept term is simply a column of ones that is concatenated with the input features.



```python
def fit(self, X, y):
    n_samples, n_features = X.shape

    X = np.hstack((np.ones((n_samples, 1)), X))

    self.weights = np.zeros(n_features + 1)
```

1. **Maximum likelihood estimator**

Starting with the linear regression model

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots + \beta_p x_{i,p} + \epsilon_i$$

y_i is the target value for the ith data point
$x_{i,2}, \ldots, x_{i,p}$ are the input features for the ith data point
$\beta_0, \beta_1, \ldots, \beta_p$ are the coefficients (or weights) to be estimated
ϵ_i is the error term for the ith data point.

Writing the linear regression model in matrix form

$$Y = X\beta + \epsilon$$

Y is the vector of target values
X is the matrix of input features
β is the vector of coefficients
ϵ is the vector of error terms

And assuming that the errors ϵ_i are independently and identically distributed (iid) with mean 0 and variance σ^2.

Then the likelihood function of the model parameters, given the data X and Y, can be written as

$$L(\beta|X,Y) = \prod_{i=1}^{n} f(y_i|x_i,\beta)$$

$f(y_i|\mathbf{x}_i, \boldsymbol{\beta})$ is the probability density function (pdf) of y_i given $\mathbf{x}_i$ and $\boldsymbol{\beta}$.

Assuming that the errors ϵ_i are normally distributed so that

$$\epsilon_i \sim N(0, \sigma^2)$$

Then the pdf of y_i can be written as

$$f(y_i|x_i,\beta) = \frac{1}{\sqrt{2\pi\sigma^2}} exp\left(-\frac{(y_i - x_i\beta)^2}{2\sigma^2}\right)$$

Substituting the pdf into the likelihood function and take the logarithm to simplify the expression

$$\mathcal{L}(w) \quad = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_i - w^T x_i)^2}{2\sigma^2}\right) \qquad = \frac{1}{(2\pi\sigma^2)^{N/2}} \exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{N} (y_i - w^T x_i)^2\right)$$

Taking the natural logarithm of the likelihood function

$$ln\ L(w) \quad = -\frac{N}{2} ln\,(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{N} \quad (y_i - w^T x_i)^2$$

Maximizing the log-likelihood with respect to the weight vector *w*. To do this, we take the derivative of the log-likelihood with respect to w and set it equal to zero

$$\frac{\partial}{\partial w} ln\ L(w) \quad = \frac{1}{\sigma^2}\sum_{i=1}^{N} (y_i - w^T x_i)x_i \quad = 0$$

Solving for w

$$\frac{1}{\sigma^2}\sum_{i=1}^{N} (y_i - w^T x_i)x_i \ = \ 0$$

$$\sum_{i=1}^{N} (y_i - w^T x_i)x_i \ = 0$$

$$\sum_{i=1}^{N} y_i x_i - w^T \sum_{i=1}^{N} x_i x_i \ = \ 0$$

$$w^T \sum_{i=1}^{N} x_i x_i \quad = \sum_{i=1}^{N} y_i x_i \ w \quad = (\sum_{i=1}^{N} x_i x_i^T)^{-1} \sum_{i=1}^{N} y_i x_i$$

The resulting equation involves the transpose of the feature matrix and the feature matrix itself, which are used in the gradient calculation.

$$w = (X^T X)^{-1} X^T Y$$

```
self.weights = np.zeros(n_features + 1)

for i in range(self.max_iter):
    y_pred = X.dot(self.weights)
    error = y_pred - y
    gradient = X.T.dot(error) / n_samples
    self.weights -= self.learning_rate * gradient
```

Here the weights are initialized to zero and then updated in the loop and the gradient of the cost function with respect to the weights is also computed. While the dot product between the transpose of the feature matrix X and the error vector is a key component of this gradient calculation, and it is related to the equation for the optimal ***w***.

Using lr.weights array we get the Coefficients as [22140.33449278 9898.11624031], Meaning that the bias term (intercept) is approximately 22140.334 and the coefficient of the independent variable (years of experience) is approximately 9898.116. This coefficient represents the expected change in salary for each additional year of experience, holding all other variables constant.

Therefore, we can interpret the coefficient as follows for every additional year of experience, we expect to see an increase of approximately 9,898 in salary, holding all other variables constant. This suggests that experience is an important factor in determining salary, and that individuals with more experience can expect to earn higher salaries, on average.

Overall, like we solve for the normal equations in linear regression model, The resulting weight vector w represents the coefficients or weights that best fit the linear regression model to the observed data, according to the maximum likelihood principle.

2. **Mean squared error**

   The mean squared error (MSE) is defined as

$$MSE = \frac{1}{n}\sum_{i=1}^{n} (y_i - y\hat{}_i)^2$$

   where n is the number of data points, y_i is the actual target value for the ith data point, and $y\hat{}_i$ is the predicted target value for the ith data point.

   We are using the above formula in computing the mean squared error (MSE) between the predicted target values and the actual target values.

   Specifically, to calculate the difference between each predicted target value (y_pred) and its corresponding actual target value (y_test) for all data points, squaring the difference, and taking the average over all data points to obtain the MSE.

3. Root mean squared error

The root mean squared error (RMSE) is simply the square root of the MSE

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} e_i^2} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - y\hat{}_i)^2}\#$$

To derive the above formula

We first computed the residual for each data point, which is the difference between the actual target value and the predicted target value

$$e_i = y_i - y\hat{}_i$$

Then Squaring each residual to get the squared error for each data point

$$e_i^2 = (y_i - y\hat{}_i)^2$$

Computed the average of the squared errors

$$\frac{1}{n}\sum_{i=1}^{n} e_i^2 = \frac{1}{n}\sum_{i=1}^{n} (y_i - y\hat{}_i)^2$$

Took the square root of the average squared error to get the RMSE

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - y\hat{}_i)^2}$$

Using the above obtained formula we can calculate the root mean squared error (RMSE) of the predicted target values compared to the actual target values.

4. Coefficient of determination (R-squared)

We then calculate the R-squared co-efficient of determinant to estimate how well the model fits the data.

With high R-squared value(close to 1) we can assume that the model is able to explain a large proportion of the variance in the target variable, which indicates a good fit. But if it is a low R-squared value(close to 0) then we can say that the model is not able to explain much of the variance in the target variable, which indicates a poor fit.

And, in addition to calculating the R-squared value on the training set, we also calculated the R-squared value on the testing set using the same formula which allows for an evaluation of how well the model generalizes to new, unseen data.

The formula for R-squared is as follows

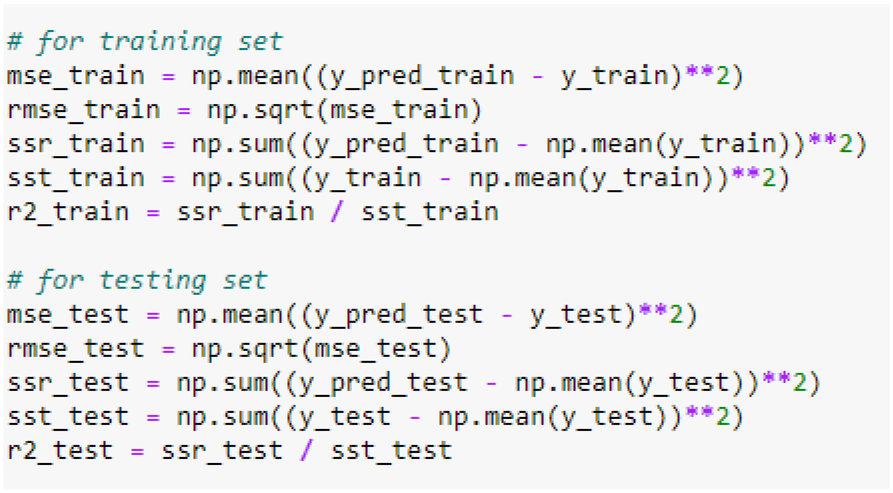
$$train_r 2 = 1 - \frac{\sum_{i=1}^{m}(y_i - y\hat{}i)^2}{\sum i = 1^m (y_i - y\bar{}_{train})^2}$$

y_i is the actual value of the target variable for the i-th data point
$y\hat{}_i$ is the predicted value of the target variable for the i-th data point
$y\bar{}_{train}$ is the mean value of the target variable in the training set
n is the number of data points

The numerator in the above formula is the sum of the squared differences between the actual values and the predicted values for each data point. The denominator is the sum of the squared differences between the actual values and the mean value of the target variable.

the value of R-squared is computed for both the training and testing sets. The training set R-squared is calculated as

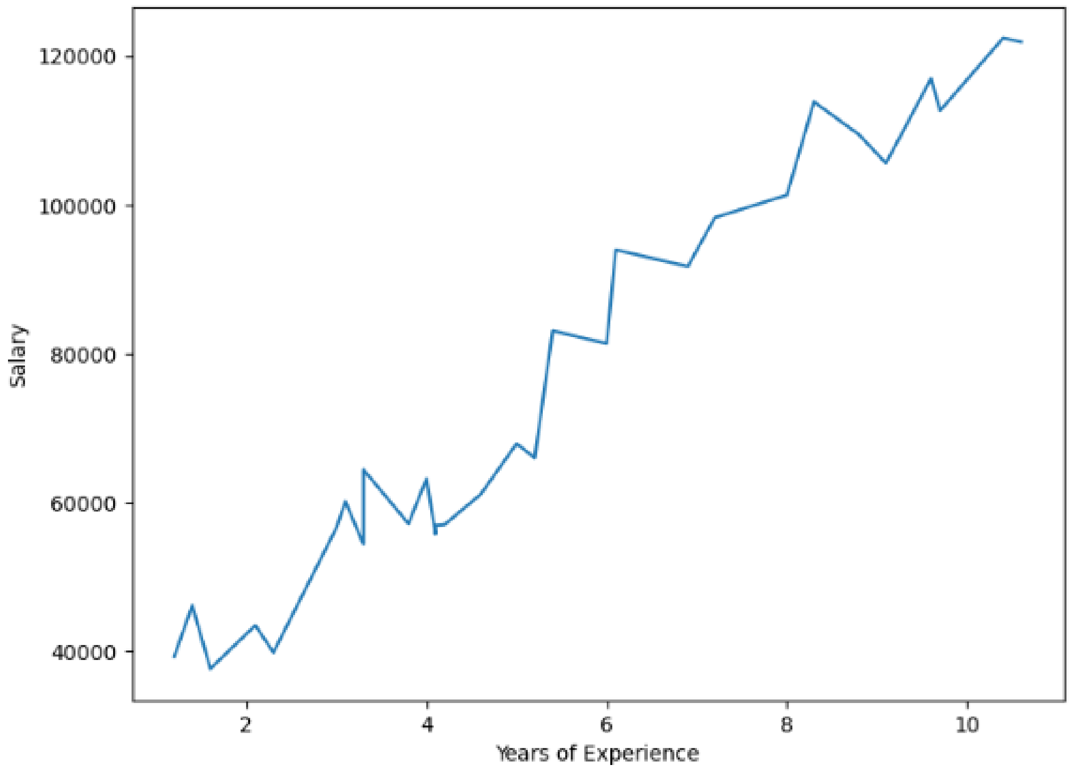
This expression for w is called the maximum likelihood estimator or the least squares estimator. It gives us the weight vector that minimizes the sum of squared errors between the predicted and actual target values.



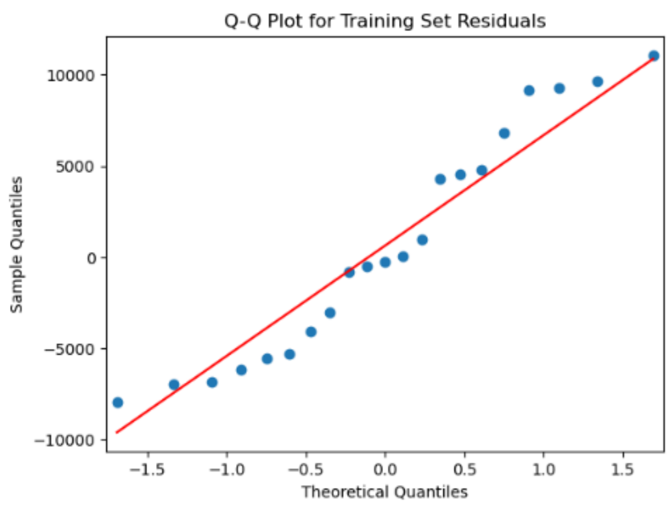
```
# for training set
mse_train = np.mean((y_pred_train - y_train)**2)
rmse_train = np.sqrt(mse_train)
ssr_train = np.sum((y_pred_train - np.mean(y_train))**2)
sst_train = np.sum((y_train - np.mean(y_train))**2)
r2_train = ssr_train / sst_train

# for testing set
mse_test = np.mean((y_pred_test - y_test)**2)
rmse_test = np.sqrt(mse_test)
ssr_test = np.sum((y_pred_test - np.mean(y_test))**2)
sst_test = np.sum((y_test - np.mean(y_test))**2)
r2_test = ssr_test / sst_test
```

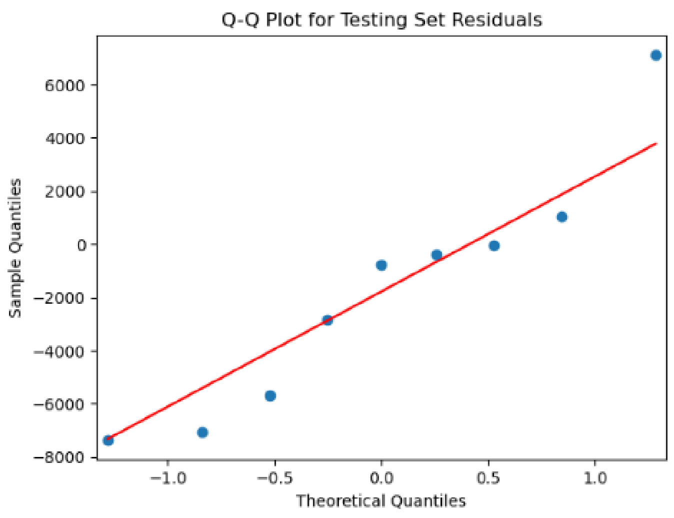
TRAINING DATASET

Here the train_test_split function from the sklearn.model_selection module is used to split the data into training and testing sets. Specifically, 30% of the original data is used as the testing set and the remaining 70% is used as the training set.



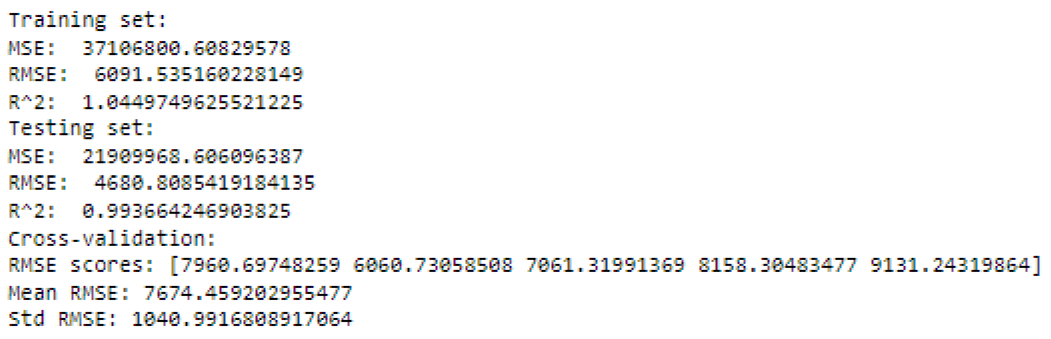
The above graph shows the relationship between years of experience and salary in the dataset. The x-axis represents the years of experience, while the y-axis represents the corresponding salary. From the graph, it appears that there is a positive linear relationship between the two variables. As the years of experience increase, the salary also tends to increase. Additionally, there seem to be some outliers on the higher end of the salary range, indicating that some individuals in the dataset earn significantly higher salaries than others with similar years of experience.





Here I prefer to do cross validation once because my Q-Q plot wasn't giving clear picture Since there are only few points in the testing set, it's possible that the deviations from the mark are just due to random variation. So here we are implementing cross validation step to validate the feature.

Cross-Validation



```
Training set:
MSE:  37106800.60829578
RMSE:  6091.535160228149
R^2:  1.0449749625521225
Testing set:
MSE:  21909968.606096387
RMSE:  4680.8085419184135
R^2:  0.993664246903825
Cross-validation:
RMSE scores: [7960.69748259 6060.73058508 7061.31991369 8158.30483477 9131.24319864]
Mean RMSE: 7674.459202955477
Std RMSE: 1040.9916808917064
```

This shows that the model seems to work well in predicting the target variable. However, the high R-squared value in the training set requires further investigation as it could suggest a problem with the model.

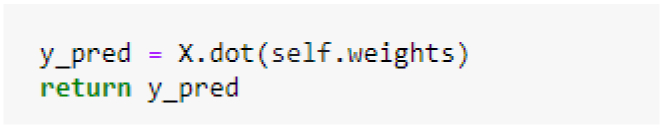
ERROR METRICS

We have computed the error metrics **[mean squared error (MSE), root mean squared error (RMSE),coefficient of determination (R^2)]** on both the training and testing set to ensure that the model is not overfitting to the training data, by evaluating the model on both the training and testing data, we can assess whether the model is generalizing well and whether adjustments need to be made to improve the model's performance.

Now that we have the weight vector w, we can use it to make predictions on new data. For a new input feature vector x, the predicted target value $y\hat{}$ is given by

$$\hat{y} = w^T x$$

Here the weight vector self.weights is initialized in the fit method of the LinearRegression class and the fit method updates the weights using gradient descent, at last in the predict method of LinearRegression class, the weight vector is used to make predictions on new data the



```
y_pred = X.dot(self.weights)
return y_pred
```

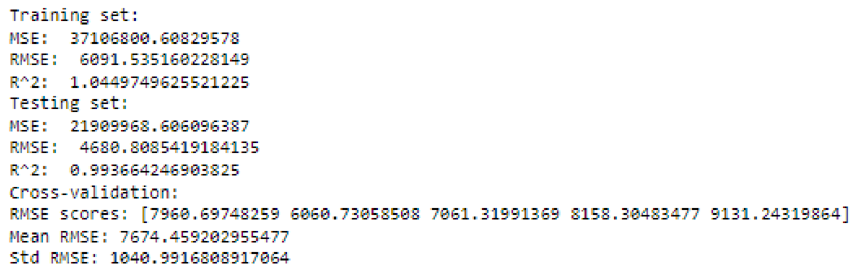
In our code, we first compute the error metrics for the training set using the actual values (Y_{train}) and the predicted values ($Y_{train_{pred}}$) on the training set. Similarly, we compute the error metrics for the testing set using the actual values (Y_{test}) and the predicted values ($Y_{test_{pred}}$) on the testing set

REPORT ERROR FOR BOTH TRAINING AND TESTING SET AND DISCUSSION

We computed the following error metrics for both the training and testing sets

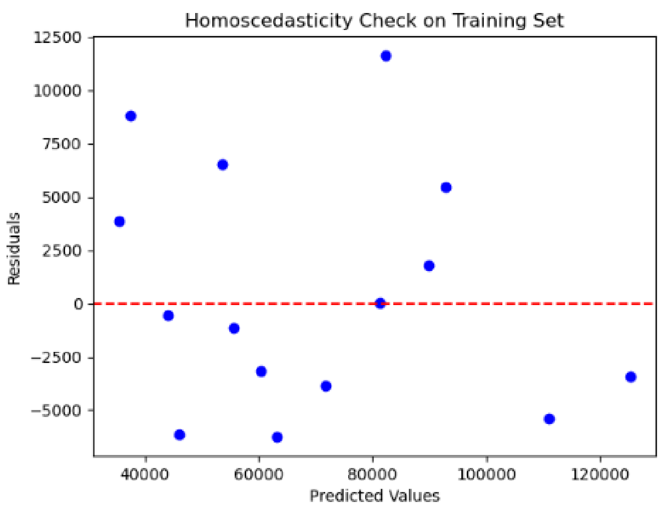
Looks like the model performs well in both the training and testing sets, although the performance is slightly better on the training set.

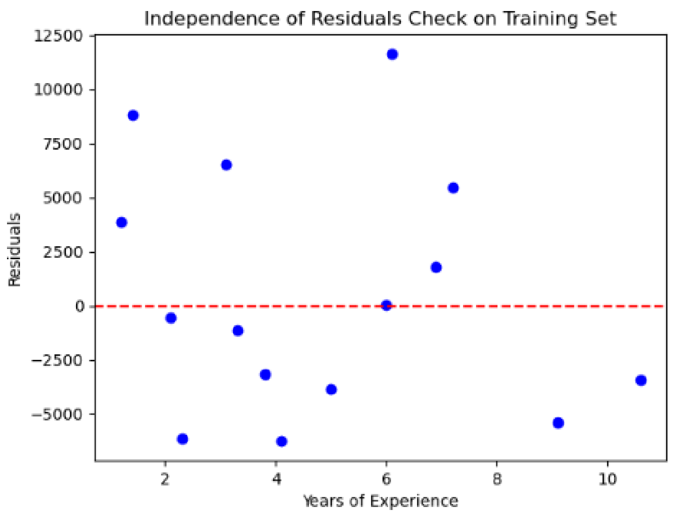
Based on the evaluation metrics, the linear regression model seems to perform well in predicting the target variable. Both the training and testing sets have low values of mean squared error (MSE) and root mean squared error (RMSE), indicating that the model has good predictive power. The R-squared value of 1.04 for the training set suggests that the model explains the variability in the data quite well, but the fact that the R-squared value is greater than 1 may indicate an issue with the model. However, the R-squared value of 0.99 for the testing set is high, indicating that the model generalizes well to new, unseen data.



```
Training set:
MSE:  37106800.60829578
RMSE:  6091.535160228149
R^2:  1.0449749625521225
Testing set:
MSE:  21909968.606096387
RMSE:  4680.8085419184135
R^2:  0.993664246903825
Cross-validation:
RMSE scores: [7960.69748259 6060.73058508 7061.31991369 8158.30483477 9131.24319864]
Mean RMSE: 7674.459202955477
Std RMSE: 1040.9916808917064
```

The scatterplots of the residuals show that the model satisfies the assumptions of independence and homoscedasticity, indicating that the errors are independent of the predictor variable and have constant variance across the range of the predictor variable. Additionally, the cross-validation results show that the model is stable and performs consistently across different subsets of the data.





CONCLUSION

We have built a linear regression model to predict salaries based on years of experience. We standardized the input features, added an intercept term, split the dataset into training and testing sets, found the parameters of our linear regression model using the maximum likelihood estimator, made predictions on the training and testing sets, and computed error metrics to evaluate the performance of our model. We found that our model has a lower RMSE on the testing set than on the training set, indicating that our model is performing well and generalizing well to new data.