Homework 1

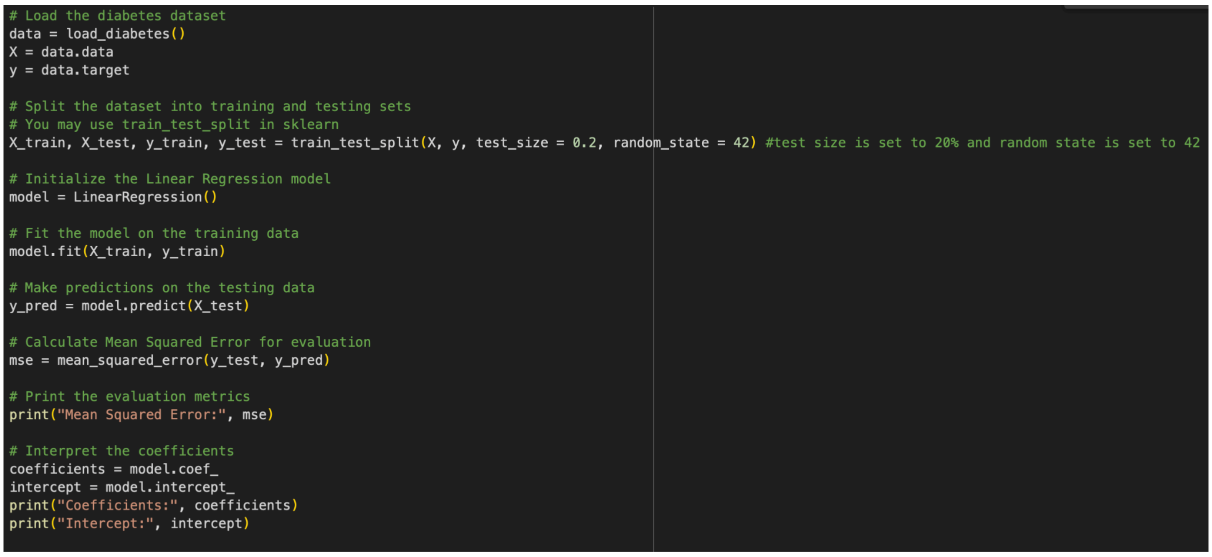
Name: Syeda Mishra Saiara
uID: u1457424

Assignment 1: Linear Regression

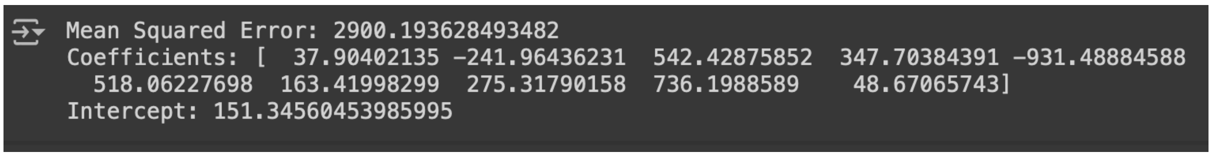
The goal of this assignment was to build a linear regression model for predicting the diabetes progression from the given diatebets database.

At first the database was load by the code given with the assignment. Then the database was split to 20% test data which automatically determines the training dataset to be 80%. Then linear regression model is set and fit into the training dataset. y_pred is the predictions on the test dataset. After the predictions are done mse is calculated to determine how good is the prediction on the test data. The mse is calculated and printed afterwards along with the coefficients and intercept.

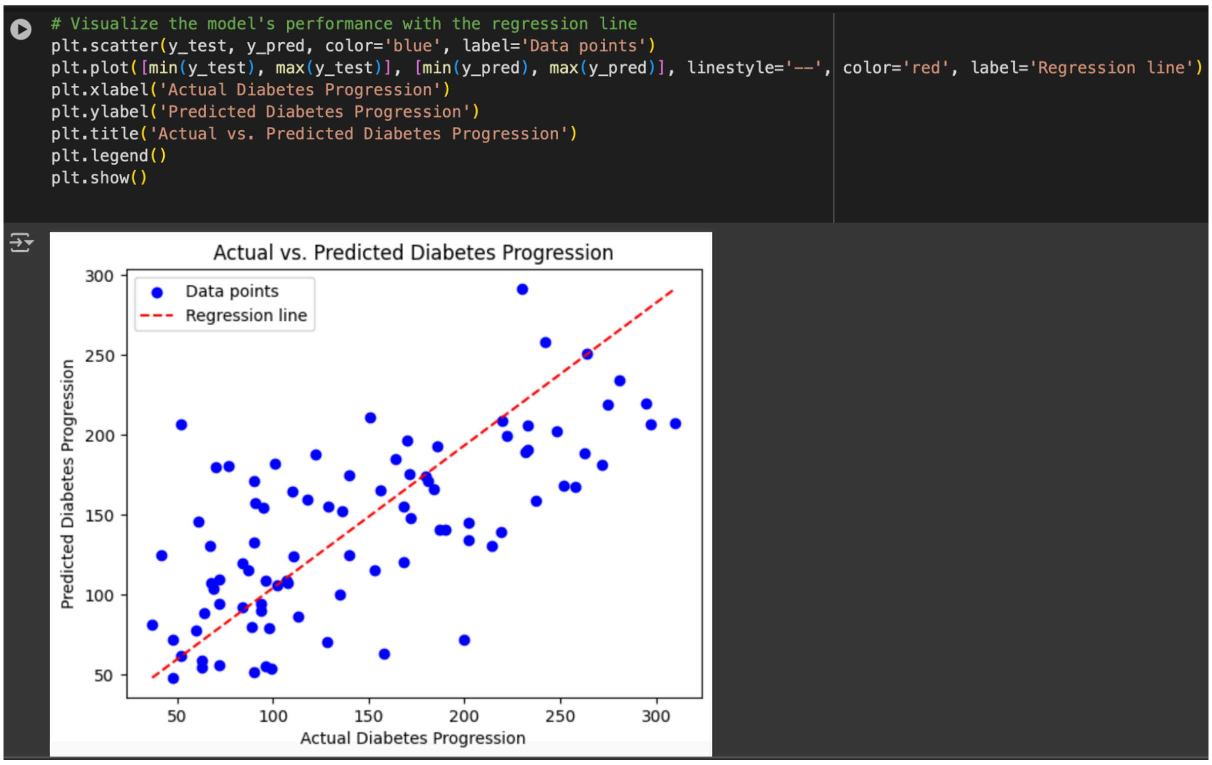
Afterwards, the models data visualizations was shown by keeping the actual data points on x axis and the predicted progression on y axis.



```python
# Load the diabetes dataset
data = load_diabetes()
X = data.data
y = data.target

# Split the dataset into training and testing sets
# You may use train_test_split in sklearn
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42) #test size is set to 20% and random state is set to 42

# Initialize the Linear Regression model
model = LinearRegression()

# Fit the model on the training data
model.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = model.predict(X_test)

# Calculate Mean Squared Error for evaluation
mse = mean_squared_error(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error:", mse)

# Interpret the coefficients
coefficients = model.coef_
intercept = model.intercept_
print("Coefficients:", coefficients)
print("Intercept:", intercept)
```



```
Mean Squared Error: 2900.193628493482
Coefficients: [  37.90402135 -241.96436231  542.42875852  347.70384391 -931.48884588
   518.06227698  163.41998299  275.31790158  736.1988589    48.67065743]
Intercept: 151.34560453985995
```



```
# Visualize the model's performance with the regression line
plt.scatter(y_test, y_pred, color='blue', label='Data points')
plt.plot([min(y_test), max(y_test)], [min(y_pred), max(y_pred)], linestyle='--', color='red', label='Regression line')
plt.xlabel('Actual Diabetes Progression')
plt.ylabel('Predicted Diabetes Progression')
plt.title('Actual vs. Predicted Diabetes Progression')
plt.legend()
plt.show()
```

Actual vs. Predicted Diabetes Progression
Data points
Regression line
Predicted Diabetes Progression
Actual Diabetes Progression
50
100
150
200
250
300

Explanations:

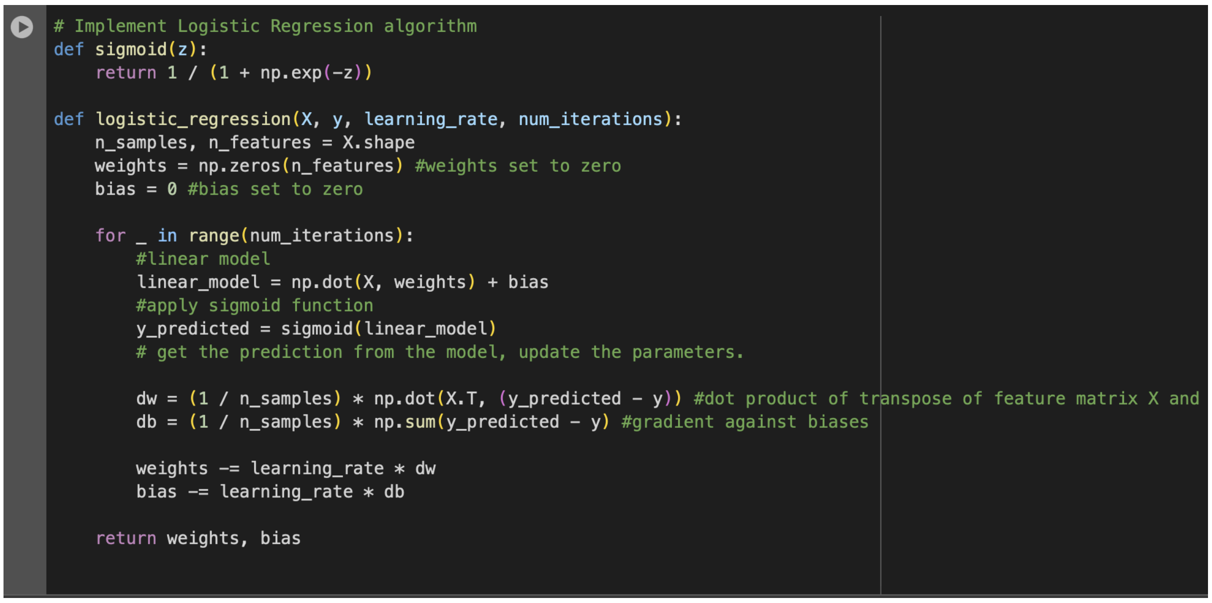
MSE determines how well the model is working to predict the results and match the actual data values. The MSE in this case is around **2900** which means it has still room to be good in predictions and has some errors in doing so.
Coefficients determines the weights on each feature or relationship between the features and predictions in the data set. Here, the positive coefficients mean that per unit increase of those features also results the predicted diabetes progression to increase. And the negatice coefficients means that increase per unit of that feature will result the predicted diabetes progression to decrease.
Intercept is the constant value if all the features are set to value 0.

For the visualizations, the comparison is done with the y_test data points and the y_pred data points to determine the model's performance. Each data points represents an actual data from test set y_test against the predicted data point y_pred. According to the visuals of this model, the data points are moderately clustered around the regression line, with a few loosely lying outliers. That means the model is performing at average and has scopes to improve the predictions.

Assignment 2: Logistic Regression

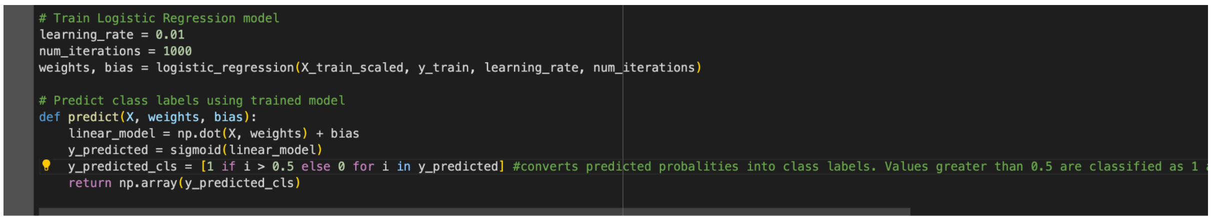
The logistic regression is performed on a breast cancer dataset provided with the assignment.

At first the weights and bias are set to zero. The sigmoid function is also defined, which is used to predict the probability of a binary value. The function logistic_regression takes the parameters feature matrix X, target values y, and learning rate and iteration numbers. The weights and bias are updated through iterations. The raw predictions from the linear model is used for getting the predicted probabilities from the sigmoid function. dw and db are calculated gradients against weights and biases respectively.



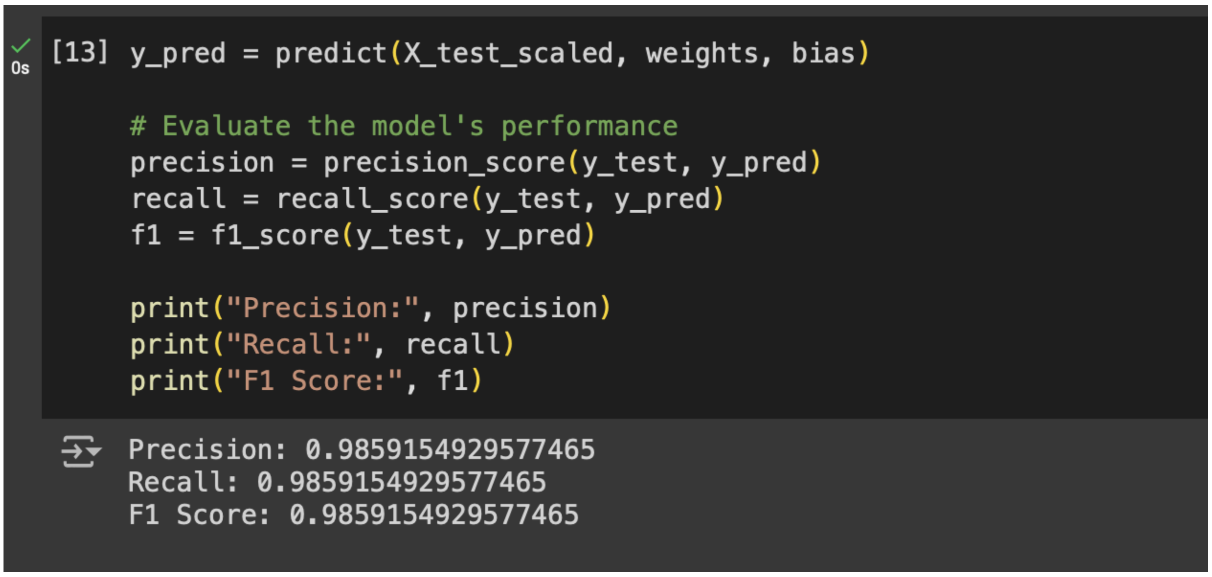
```
# Implement Logistic Regression algorithm
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def logistic_regression(X, y, learning_rate, num_iterations):
    n_samples, n_features = X.shape
    weights = np.zeros(n_features) #weights set to zero
    bias = 0 #bias set to zero

    for _ in range(num_iterations):
        #linear model
        linear_model = np.dot(X, weights) + bias
        #apply sigmoid function
        y_predicted = sigmoid(linear_model)
        # get the prediction from the model, update the parameters.

        dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y)) #dot product of transpose of feature matrix X and
        db = (1 / n_samples) * np.sum(y_predicted - y) #gradient against biases

        weights -= learning_rate * dw
        bias -= learning_rate * db

    return weights, bias
```

Then the model is trained on normalized feature data (X) and target labels (y). The function predict() works as a prediction function of binary values based on the threshold 0.5.



```
# Train Logistic Regression model
learning_rate = 0.01
num_iterations = 1000
weights, bias = logistic_regression(X_train_scaled, y_train, learning_rate, num_iterations)

# Predict class labels using trained model
def predict(X, weights, bias):
    linear_model = np.dot(X, weights) + bias
    y_predicted = sigmoid(linear_model)
    y_predicted_cls = [1 if i > 0.5 else 0 for i in y_predicted] #converts predicted probalities into class labels. Values greater than 0.5 are classified as 1
    return np.array(y_predicted_cls)
```

The model is then evaluated using precision, recall and F1 score.



```
[13] y_pred = predict(X_test_scaled, weights, bias)

     # Evaluate the model's performance
     precision = precision_score(y_test, y_pred)
     recall = recall_score(y_test, y_pred)
     f1 = f1_score(y_test, y_pred)

     print("Precision:", precision)
     print("Recall:", recall)
     print("F1 Score:", f1)
```

```
Precision: 0.9859154929577465
Recall: 0.9859154929577465
F1 Score: 0.9859154929577465
```

Explanation:

Precision is the ratio of true positive predictions to the total number of predictions made by the model. Here precision is ~ 0.985, which means around 98.59% predictions are true implying that there are very few false positives.

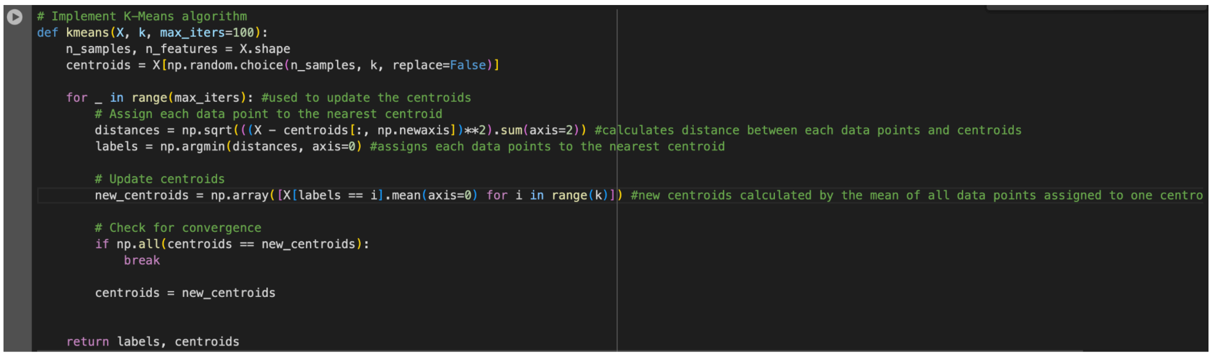
Recall is the ratio of true positive predictions to the total number of actual positive predictions made. Here, the recall is ~0.985, which means around 98.5% of the actual positive cases were identified by the model.

F1 score is the harmonic mean of precision and recall. 0.9859 F1 score indicates a good balance between recall and precision.

Assignment 3: K- means

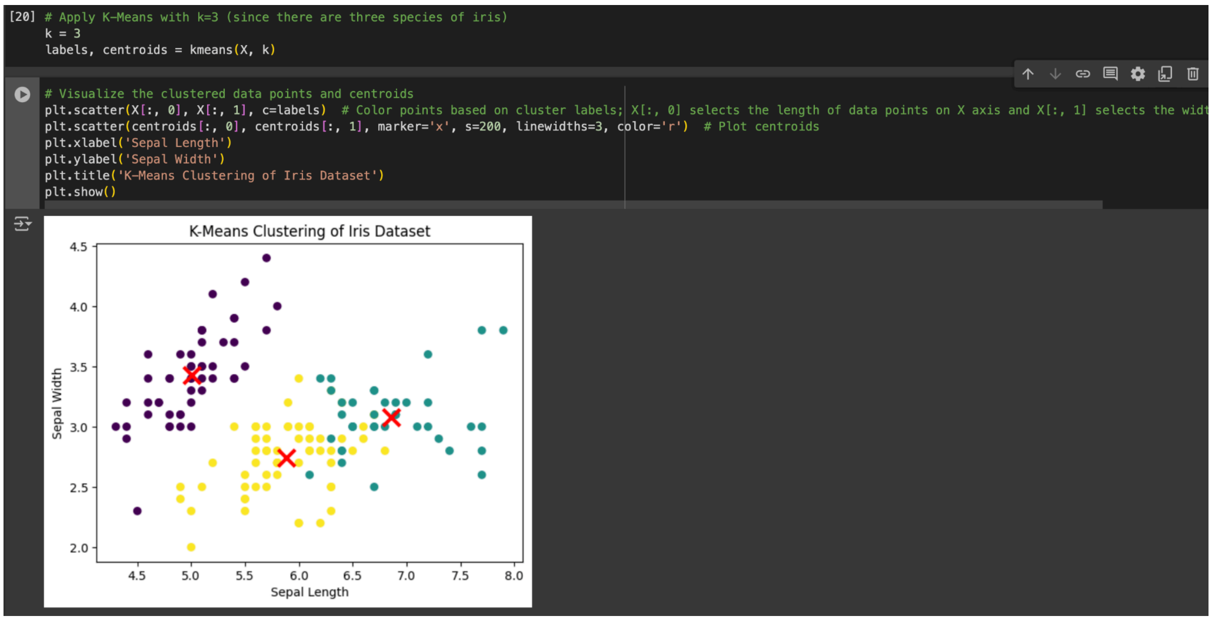
The goal of this assignment is to implement the K - means algorithm by the provided iris dataset. K means is an unsupervised algorith used for clustering data points into groups.

The function kmeans() first retrieves the number of samples and features from the shape of the feature matrix X. And then the centroids are initialized randomly.
After that, the centroids are updated through a loop of iterations. To assign the data points to a centroid, firstly the distance between each data points and centroids are calculated. Then each data points are assigned to the nearest centroids. Centroids are updated by the mean of all data points assigned to the centroids.
Then if centroids are changed from the previous iterations the loop breaks and the new centroids with labels are returned.



```python
# Implement K-Means algorithm
def kmeans(X, k, max_iters=100):
    n_samples, n_features = X.shape
    centroids = X[np.random.choice(n_samples, k, replace=False)]

    for _ in range(max_iters): #used to update the centroids
        # Assign each data point to the nearest centroid
        distances = np.sqrt(((X - centroids[:, np.newaxis])**2).sum(axis=2)) #calculates distance between each data points and centroids
        labels = np.argmin(distances, axis=0) #assigns each data points to the nearest centroid

        # Update centroids
        new_centroids = np.array([X[labels == i].mean(axis=0) for i in range(k)]) #new centroids calculated by the mean of all data points assigned to one centro

        # Check for convergence
        if np.all(centroids == new_centroids):
            break

        centroids = new_centroids


    return labels, centroids
```

Then the algorithm is applied to the iris dataset with k= 3. Then the visualization is implemented.



```
[20] # Apply K-Means with k=3 (since there are three species of iris)
     k = 3
     labels, centroids = kmeans(X, k)
```

```
# Visualize the clustered data points and centroids
plt.scatter(X[:, 0], X[:, 1], c=labels)  # Color points based on cluster labels; X[:, 0] selects the length of data points on X axis and X[:, 1] selects the widt
plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', s=200, linewidths=3, color='r')  # Plot centroids
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('K-Means Clustering of Iris Dataset')
plt.show()
```

Explanation:

The data points were clustered into three groups as shown.