Testing and Coding: Deliverable 3

Yukta, Brij, Mayank and Nimish

Code Base

Linear Regression (LinearRegression class):

Class Structure:

1. *Encapsulation of linear regression functionalities.*
2. *Key attributes: `train_points`, `test_points`, `avg_x`, `avg_y`, etc.*

Modular Methods:

1. *`calcMeans`: Calculates means of training data points.*
2. `calcStandardDeviation`: Computes standard deviations of variables.
3. `calcCorrelation`: Calculates coefficient of correlation and linear model parameters.
4. `train_test_split`: Splits data into training and testing sets.
5. `predict`: Predicts dependent variable based on linear model.
6. `calcMeanSquaredError` and `calcRSquareScore`: Calculate error metrics.

Main Method:

1. *Instance of `LinearRegression` created.*
2. *Data loaded from CSV file (`testLR.csv`).*
3. *`fit` method called to train linear regression model.*
4. *Display results: correlation coefficients, linear model equation, mean squared error.*
5. *Logistic regression classification hardcoded.*

Logistic Regression (LogisticRegression class):

Class Structure:

1. *Encapsulation of logistic regression functionalities.*
2. *Key attributes: `weights`, `learningRate`.*

Modular Methods:

1. *`sigmoid`: Calculates sigmoid function for logistic regression.*

2. *`predict`: Predicts probability of a positive class.*
3. *`updateWeights`: Updates weights during training.*
4. *`train`: Trains logistic regression model.*

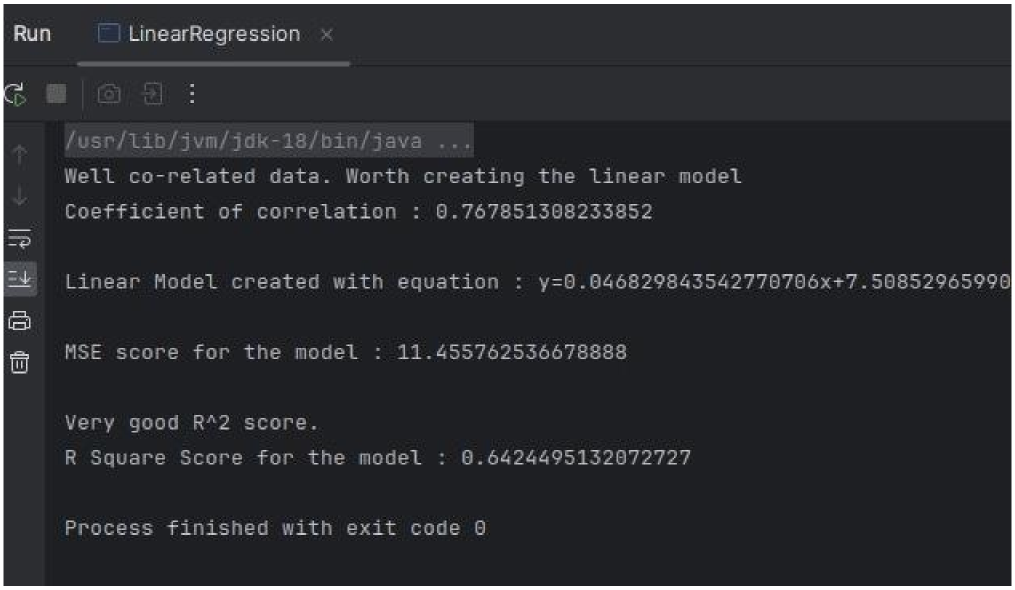
Main Method:

1. ***I**nstance of `LogisticRegression` created.*
2. *Data loaded from CSV file (`test.csv`).*
3. *Logistic regression model trained using `train` method.*
4. *Sample test data used for predictions.*
5. *Display results.*

Sample Run:

Linear Regression:

1. *Load data from `testLR.csv`.*
2. *Train linear regression model.*
3. *Display correlation coefficients, linear model equation, mean squared error.*



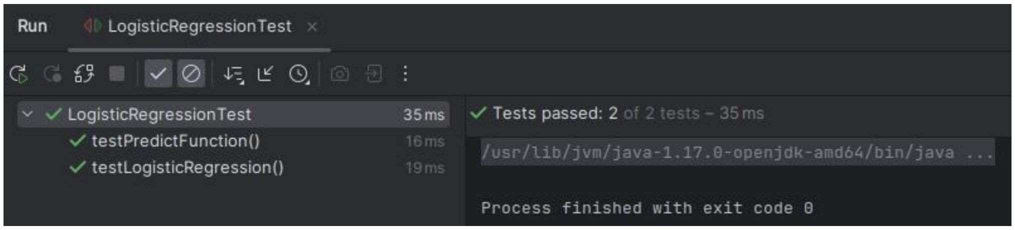
```
/usr/lib/jvm/jdk-18/bin/java ...
Well co-related data. Worth creating the linear model
Coefficient of correlation : 0.767851308233852

Linear Model created with equation : y=0.04682984354277070６x+7.50852965990
```

```
MSE score for the model : 11.455762536678888

Very good R^2 score.
R Square Score for the model : 0.6424495132072727

Process finished with exit code 0
```

Logistic Regression:

1. *Load data from `test.csv`.*
2. *Train logistic regression model.*

3. *Display predictions for sample test data.*



```
✓ LogisticRegressionTest        35 ms
    ✓ testPredictFunction()     16 ms
    ✓ testLogisticRegression()  19 ms
```

```
✓ Tests passed: 2 of 2 tests – 35 ms
/usr/lib/jvm/java-1.17.0-openjdk-amd64/bin/java ...

Process finished with exit code 0
```

FRs and NFRs Implementation :

Reliability:

1. Accurate calculations in linear and logistic regression methods.
2. Reliable training and prediction processes.

Usability:

1. User-friendly methods (`fit`, `predict`, etc.) for easy interaction.
2. Data loading from CSV for simplicity.

Maintainability:

1. Code organization into separate classes with encapsulated methods.

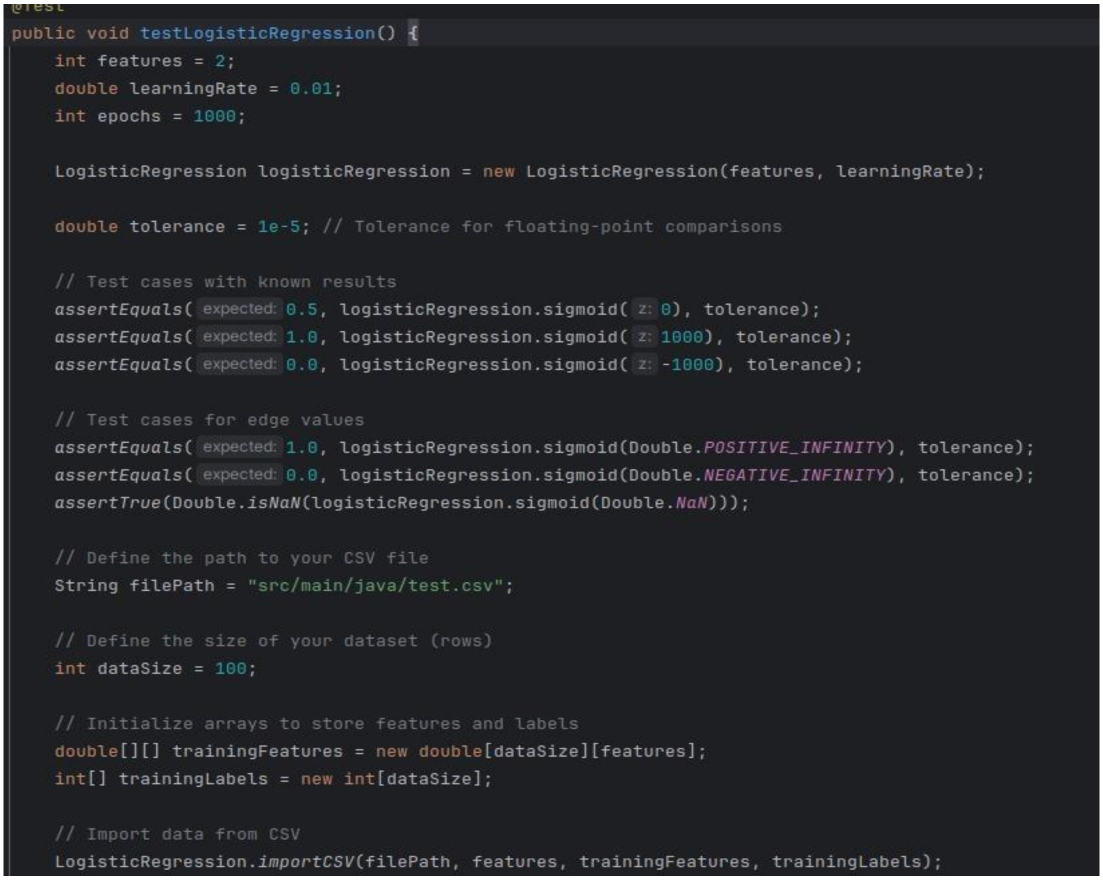
TDD (Test-Driven Development) :

1. Test classes (`LinearRegressionTest` and `LogisticRegressionTest`) for testing linear and logistic regression.
2. Utilization of JUnit framework for standardized testing.
   - testCalcMeans: Tests the calculation of means in linear regression with specific data points. testCalcStandardDeviation:
   - Checks standard deviation calculation in linear regression with specific data.
   - testCalcCorrelation: Verifies correlation calculation in linear regression with given data points.

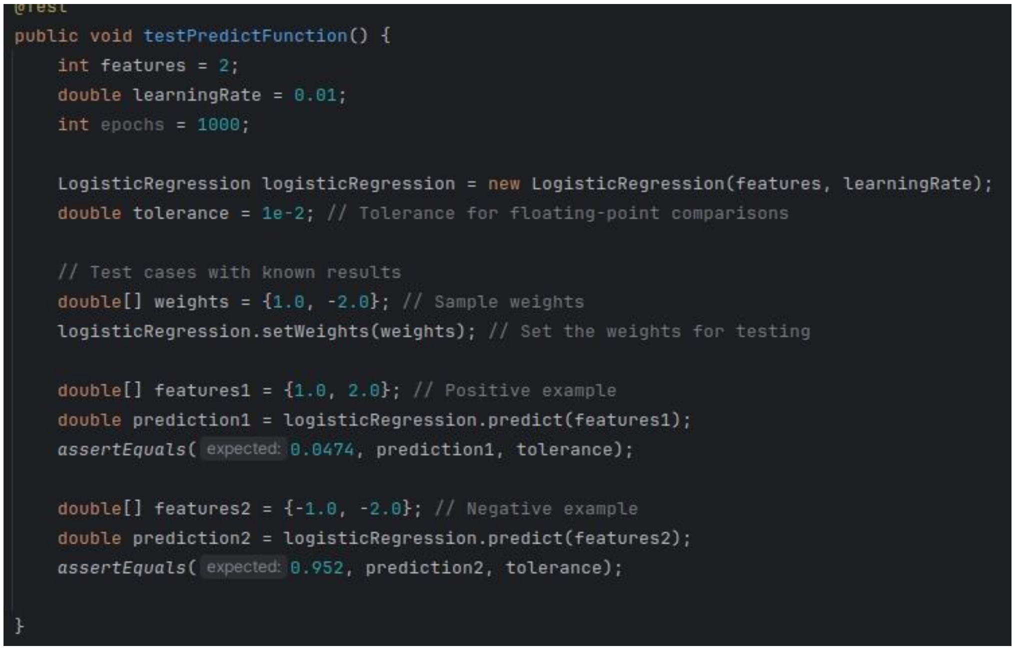
- testTrainTestSplit: Ensures correct splitting of data into training and testing sets in linear regression.
- testPredict: Validates linear regression prediction for a given slope, intercept, and input value.
- testCalcMeanSquaredError: Tests mean squared error calculation in linear regression with predefined data.
- testCalcRSquareScore: Verifies R-square score calculation in linear regression with specified values. testPredictFunction: Tests logistic regression predictions for positive and negative examples with known weights.
- testLogisticRegression: Tests various aspects of logistic regression, including the sigmoid function, CSV data import, and model training with predefined data.

ScreenShots of Working :

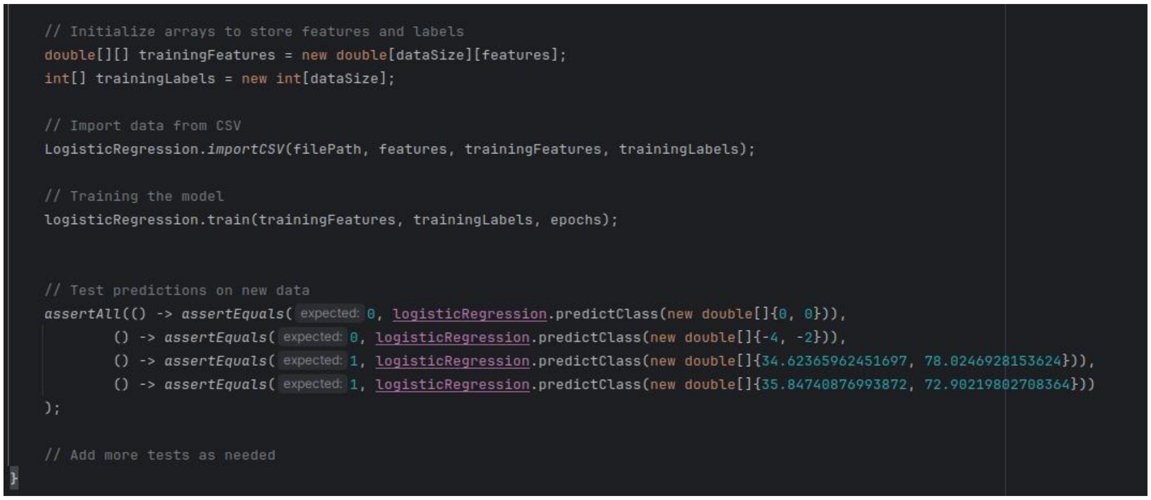
Test case Logistic:



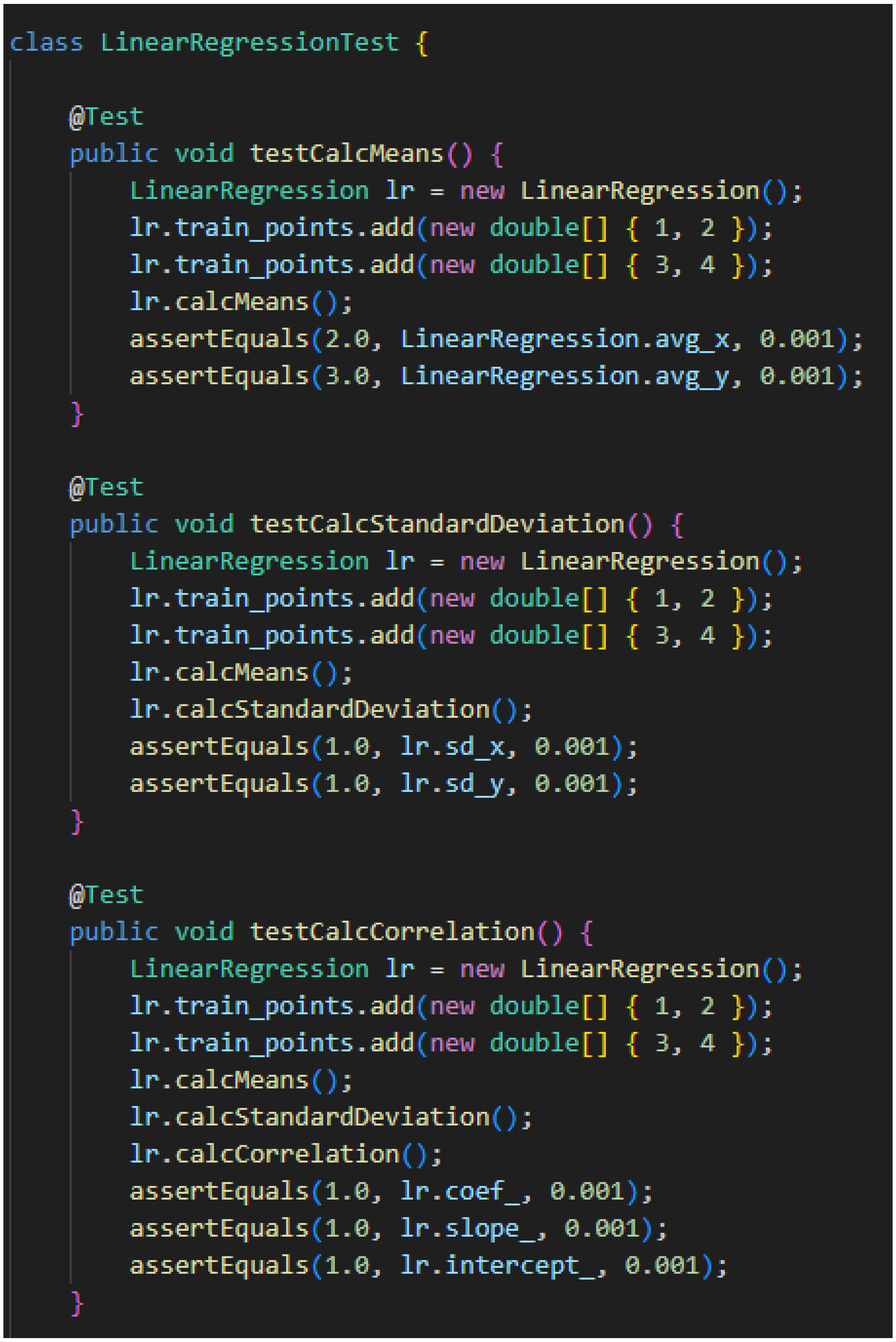
```
@Test
public void testLogisticRegression() {
    int features = 2;
    double learningRate = 0.01;
    int epochs = 1000;

    LogisticRegression logisticRegression = new LogisticRegression(features, learningRate);

    double tolerance = 1e-5; // Tolerance for floating-point comparisons

    // Test cases with known results
    assertEquals( expected: 0.5, logisticRegression.sigmoid( z: 0), tolerance);
    assertEquals( expected: 1.0, logisticRegression.sigmoid( z: 1000), tolerance);
    assertEquals( expected: 0.0, logisticRegression.sigmoid( z: -1000), tolerance);

    // Test cases for edge values
    assertEquals( expected: 1.0, logisticRegression.sigmoid(Double.POSITIVE_INFINITY), tolerance);
    assertEquals( expected: 0.0, logisticRegression.sigmoid(Double.NEGATIVE_INFINITY), tolerance);
    assertTrue(Double.isNaN(logisticRegression.sigmoid(Double.NaN)));

    // Define the path to your CSV file
    String filePath = "src/main/java/test.csv";

    // Define the size of your dataset (rows)
    int dataSize = 100;

    // Initialize arrays to store features and labels
    double[][] trainingFeatures = new double[dataSize][features];
    int[] trainingLabels = new int[dataSize];

    // Import data from CSV
    LogisticRegression.importCSV(filePath, features, trainingFeatures, trainingLabels);
```



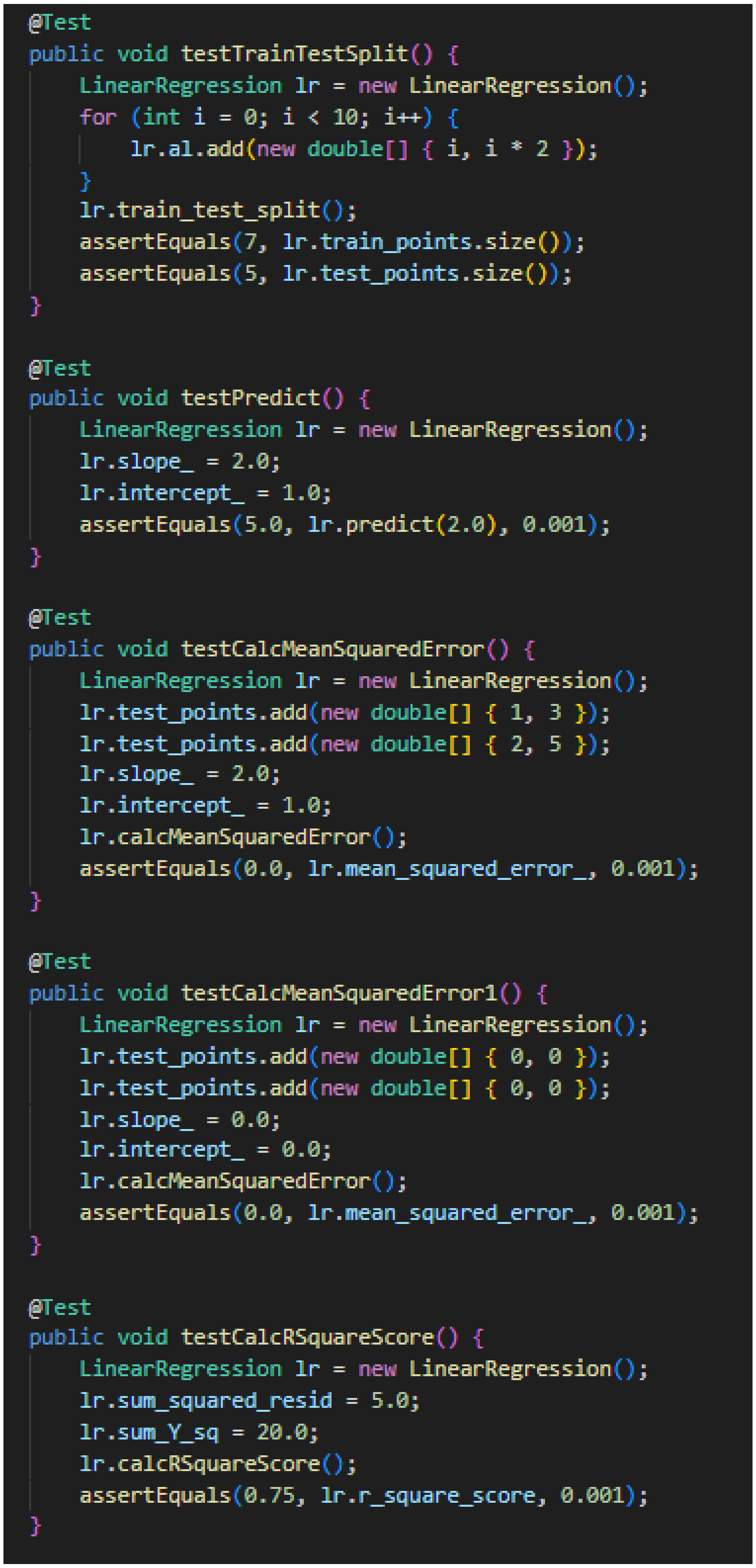
```java
@Test
public void testPredictFunction() {
    int features = 2;
    double learningRate = 0.01;
    int epochs = 1000;

    LogisticRegression logisticRegression = new LogisticRegression(features, learningRate);
    double tolerance = 1e-2; // Tolerance for floating-point comparisons

    // Test cases with known results
    double[] weights = {1.0, -2.0}; // Sample weights
    logisticRegression.setWeights(weights); // Set the weights for testing

    double[] features1 = {1.0, 2.0}; // Positive example
    double prediction1 = logisticRegression.predict(features1);
    assertEquals( expected: 0.0474, prediction1, tolerance);

    double[] features2 = {-1.0, -2.0}; // Negative example
    double prediction2 = logisticRegression.predict(features2);
    assertEquals( expected: 0.952, prediction2, tolerance);

}
```



```java
    // Initialize arrays to store features and labels
    double[][] trainingFeatures = new double[dataSize][features];
    int[] trainingLabels = new int[dataSize];

    // Import data from CSV
    LogisticRegression.importCSV(filePath, features, trainingFeatures, trainingLabels);

    // Training the model
    logisticRegression.train(trainingFeatures, trainingLabels, epochs);


    // Test predictions on new data
    assertAll(() -> assertEquals( expected: 0, logisticRegression.predictClass(new double[]{0, 0})),
            () -> assertEquals( expected: 0, logisticRegression.predictClass(new double[]{-4, -2})),
            () -> assertEquals( expected: 1, logisticRegression.predictClass(new double[]{34.62365962451697, 78.0246928153624})),
            () -> assertEquals( expected: 1, logisticRegression.predictClass(new double[]{35.84740876993872, 72.90219802708364}))
    );

    // Add more tests as needed
}
```



```
class LinearRegressionTest {

    @Test
    public void testCalcMeans() {
        LinearRegression lr = new LinearRegression();
        lr.train_points.add(new double[] { 1, 2 });
        lr.train_points.add(new double[] { 3, 4 });
        lr.calcMeans();
        assertEquals(2.0, LinearRegression.avg_x, 0.001);
        assertEquals(3.0, LinearRegression.avg_y, 0.001);
    }

    @Test
    public void testCalcStandardDeviation() {
        LinearRegression lr = new LinearRegression();
        lr.train_points.add(new double[] { 1, 2 });
        lr.train_points.add(new double[] { 3, 4 });
        lr.calcMeans();
        lr.calcStandardDeviation();
        assertEquals(1.0, lr.sd_x, 0.001);
        assertEquals(1.0, lr.sd_y, 0.001);
    }

    @Test
    public void testCalcCorrelation() {
        LinearRegression lr = new LinearRegression();
        lr.train_points.add(new double[] { 1, 2 });
        lr.train_points.add(new double[] { 3, 4 });
        lr.calcMeans();
        lr.calcStandardDeviation();
        lr.calcCorrelation();
        assertEquals(1.0, lr.coef_, 0.001);
        assertEquals(1.0, lr.slope_, 0.001);
        assertEquals(1.0, lr.intercept_, 0.001);
    }
```



```java
@Test
public void testTrainTestSplit() {
    LinearRegression lr = new LinearRegression();
    for (int i = 0; i < 10; i++) {
        lr.al.add(new double[] { i, i * 2 });
    }
    lr.train_test_split();
    assertEquals(7, lr.train_points.size());
    assertEquals(5, lr.test_points.size());
}

@Test
public void testPredict() {
    LinearRegression lr = new LinearRegression();
    lr.slope_ = 2.0;
    lr.intercept_ = 1.0;
    assertEquals(5.0, lr.predict(2.0), 0.001);
}

@Test
public void testCalcMeanSquaredError() {
    LinearRegression lr = new LinearRegression();
    lr.test_points.add(new double[] { 1, 3 });
    lr.test_points.add(new double[] { 2, 5 });
    lr.slope_ = 2.0;
    lr.intercept_ = 1.0;
    lr.calcMeanSquaredError();
    assertEquals(0.0, lr.mean_squared_error_, 0.001);
}

@Test
public void testCalcMeanSquaredError1() {
    LinearRegression lr = new LinearRegression();
    lr.test_points.add(new double[] { 0, 0 });
    lr.test_points.add(new double[] { 0, 0 });
    lr.slope_ = 0.0;
    lr.intercept_ = 0.0;
    lr.calcMeanSquaredError();
    assertEquals(0.0, lr.mean_squared_error_, 0.001);
}

@Test
public void testCalcRSquareScore() {
    LinearRegression lr = new LinearRegression();
    lr.sum_squared_resid = 5.0;
    lr.sum_Y_sq = 20.0;
    lr.calcRSquareScore();
    assertEquals(0.75, lr.r_square_score, 0.001);
}
```