1. Write a simple Scala program that prints a welcome message for data scientists.

```
// WelcomeMessage.scala

object WelcomeMessage {
  def main(args: Array[String]): Unit = {
    println("Welcome to the world of Data Science")
  }
}
```

Output:-

```
Welcome to the world of Data Science
```

2. Calculate mean, median, and mode of a list of numbers. Implement basic statistical calculations using Scala collections.

```
object Calculation {
  def main(args: Array[String]): Unit = {
    val numbers = List(1, 2, 2, 3, 4, 5, 5, 5, 6)

    // Calculate Mean
    val mean = numbers.sum.toDouble / numbers.length

    // Calculate Median
    val sorted = numbers.sorted
    val median = if (numbers.length % 2 == 0) {
      val (upper, lower) = sorted.splitAt(numbers.length / 2)
      (upper.last + lower.head).toDouble / 2
    } else {
      sorted(numbers.length / 2)
    }

    // Calculate Mode
    val grouped =
numbers.groupBy(identity).view.mapValues(_.size)
    val maxFrequency = grouped.values.max
    val mode = grouped.filter(_._2 ==
maxFrequency).keys.toList

    // Output results
```

```
    println(s"Numbers: $numbers")
    println(f"Mean:   $mean%.2f")
    println(s"Median: $median")
    println(s"Mode:   ${mode.mkString(", ")}")
  }
}
```

Output:-

```
[info] running Calculation
Numbers: List(1, 2, 2, 3, 4, 5, 5, 5, 6)
Mean:   3.67
Median: 4.0
Mode:   5
```

3. Generate a random dataset of 10 numbers and calculate its variance and standard deviation.

```
import scala.util.Random
import scala.math.sqrt

object StatsCalculator {
  def main(args: Array[String]): Unit = {
    // Generate random dataset of 10 numbers between 1 and 100
    val data = Array.fill(10)(Random.nextInt(100) + 1)

    println("Generated Dataset: " + data.mkString(", "))

    val mean = data.sum.toDouble / data.length
    val variance = data.map(x => math.pow(x - mean, 2)).sum /
data.length
    val stdDev = sqrt(variance)

    println(f"Mean: $mean%.2f")
    println(f"Variance: $variance%.2f")
    println(f"Standard Deviation: $stdDev%.2f")
  }
}
```

Output:-

```
Generated Dataset: 7, 70, 86, 96, 81, 4, 91, 79, 15, 56
Mean: 58.50
Variance: 1179.85
Standard Deviation: 34.35
```

4. Create a dense vector using Breeze and calculate its sum, mean, and dot product with another vector.

```
import breeze.linalg._
import breeze.stats._

object BreezeVectorOps {
  def main(args: Array[String]): Unit = {
    // Create two DenseVectors
    val vector1 = DenseVector(3.0, 6.0, 9.0, 12.0)
    val vector2 = DenseVector(1.0, 2.0, 3.0, 4.0)

    // Calculate sum and mean of vector1
    val sum1 = sum(vector1)
    val mean1 = mean(vector1)

    // Calculate dot product between vector1 and vector2
    val dotProd = vector1 dot vector2

    // Output the results
    println(s"Vector 1: ${vector1.toArray.mkString(", ")}")
    println(s"Vector 2: ${vector2.toArray.mkString(", ")}")
    println(s"Sum of Vector 1: $sum1")
    println(f"Mean of Vector 1: $mean1%.2f")
    println(s"Dot Product (Vector1 · Vector2): $dotProd")
  }
}
```

Output:-

```
Vector 1: 3.0, 6.0, 9.0, 12.0
Vector 2: 1.0, 2.0, 3.0, 4.0
Sum of Vector 1: 30.0
Mean of Vector 1: 7.50
Dot Product (Vector1 ∏ Vector2): 90.0
```

5. Generate a random matrix using Breeze and compute its transpose and determinant.

```
import breeze.linalg._
import breeze.stats.distributions._

object GenerateMatrix {
  def main(args: Array[String]): Unit = {
    // Provide an implicit RandBasis
    implicit val rand: RandBasis = Rand

    // Generate 3x3 random matrix (values from 1 to 10)
    val matrix: DenseMatrix[Double] = DenseMatrix.rand(3, 3,
Uniform(1, 10))

    // Transpose
    val transposed = matrix.t

    // Use a different variable name to avoid conflict with `det`
method
    val determinantValue = det(matrix)

    // Output
    println("Original Matrix:")
    println(matrix)

    println("\nTransposed Matrix:")
    println(transposed)

    println(f"\nDeterminant: $determinantValue%.4f")
  }
}
```

Output:-

```
Original Matrix:
8.740492383914969   4.835187513486994   2.275797921530007
9.97673428036809    5.442377297694735   6.793125239538336
4.0998542894383805  6.256331326362752   9.122127088158523

Transposed Matrix:
8.740492383914969  9.97673428036809   4.0998542894383805
4.835187513486994  5.442377297694735  6.256331326362752
2.275797921530007  6.793125239538336  9.122127088158523

Determinant: -151.6517
```

6. Slice a Breeze matrix to extract a sub-matrix and calculate its row and column sums.

```
import breeze.linalg._
import breeze.stats.distributions._

object RowsAndColsSUM {
  def main(args: Array[String]): Unit = {
    // Generate a 4x4 random matrix
    implicit val rand: RandBasis = Rand
    val matrix = DenseMatrix.rand(4, 4, Uniform(1, 10))

    println("Original 4x4 Matrix:")
    println(matrix)

    // Slice a 2x3 submatrix from top-left corner (rows 0-1, cols 0-2)
    val subMatrix = matrix(0 to 1, 0 to 2)

    println("\nExtracted Submatrix (2x3 from top-left):")
    println(subMatrix)

    // Row sums
    val rowSums = sum(subMatrix(*, ::)) // Sum across columns for
each row

    // Column sums
    val colSums = sum(subMatrix(::, *)) // Sum down rows for each
column

    println("\nRow Sums:")
    println(rowSums)

    println("\nColumn Sums:")
    println(colSums)
  }
}
```

Output:-

```
Original 4x4 Matrix:
8.023768507054218   7.920722458653293  5.3291365090146705  9.272462797019594
4.1462803892120625  2.745957298887234  1.8234797903737987  8.758358037662267
3.265534381008216   4.161010386695349  7.939596687354741   8.486127067502778
9.618745468686527   7.111952314901663  1.0958221490375282  5.982860753045352

Extracted Submatrix (2x3 from top-left):
8.023768507054218   7.920722458653293  5.3291365090146705
4.1462803892120625  2.745957298887234  1.8234797903737987

Row Sums:
DenseVector(21.273627474722183, 8.715717478473096)

Column Sums:
Transpose(DenseVector(12.17004889626628, 10.666679757540527, 7.15261629938847))
```

7. Write a program to perform element-wise addition, subtraction, multiplication, and division of two Breeze matrices.

```
import breeze.linalg._
import breeze.numerics._
import breeze.stats.distributions._

object MatrixElementwiseOps {
  def main(args: Array[String]): Unit = {
    // Required for random generation
    implicit val rand: RandBasis = Rand

    // Create two 3x3 random matrices
    val matA = DenseMatrix.rand(3, 3, Uniform(1, 10))
    val matB = DenseMatrix.rand(3, 3, Uniform(1, 10))

    println("Matrix A:")
    println(matA)

    println("\nMatrix B:")
    println(matB)

    // Element-wise operations
    val added = matA + matB
    val subtracted = matA - matB
    val multiplied = matA.mapPairs { case ((i, j), v) => v * matB(i, j) }
// manually element-wise
    val divided = matA.mapPairs { case ((i, j), v) => v / matB(i, j) }      //
manually element-wise
```

```
    println("\nElement-wise Addition (A + B):")
    println(added)

    println("\nElement-wise Subtraction (A - B):")
    println(subtracted)

    println("\nElement-wise Multiplication (A .* B):")
    println(multiplied)

    println("\nElement-wise Division (A ./ B):")
    println(divided)
  }
}
```

Output:-

```
Matrix A:
6.975416478145515  6.414193119801682   9.191814163629576
2.583229094164266  7.221360812782065   7.739993952428753
5.381306407774942  5.6235067636688045  4.903221896563601

Matrix B:
9.657124618895256  4.431457903466795  2.7691535582782074
5.980003884162247  8.132854404298714  4.300703399859345
6.13254521661325   4.773837365788202  3.4962742709152

Element-wise Addition (A + B):
16.63254109704077   10.845651023268477  11.960967721907783
8.563232978326514   15.354215217080778  12.040697352288099
11.513851624388192  10.397344129457007  8.399496167478802

Element-wise Subtraction (A - B):
-2.681708140749741   1.9827352163348877   6.4226606053513695
-3.3967747899979814  -0.9114935915166482  3.4392905525694077
-0.7512388088383082  0.8496693978806027   1.4069476256484013

Element-wise Multiplication (A .* B):
67.3624661981467    28.4242267951075    25.453544898246868
15.447720016783235  58.73027609126476   33.28741830610111
33.001104870130455  26.845706715164823  17.14300856154335

Element-wise Division (A ./ B):
0.7223078041777906  1.44742278038651    3.319358775229794
0.43197782881142    0.8879245162639494  1.7997041955234323
0.8774996706419417  1.177984571483263   1.4024134025618398
```

8. Calculate the moving average of a time series data using Scala collections.

```
object MovingAverage {
  def main(args: Array[String]): Unit = {
    // Time series data (example)
    val timeSeries = List(10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0)

    // Define the window size (e.g., 3-point moving average)
    val windowSize = 3

    // Calculate moving average
    val movingAverages = timeSeries.sliding(windowSize).map(window
=> window.sum / window.size).toList

    // Print results
    println(s"Original Time Series: $timeSeries")
    println(s"Moving Average (window size = $windowSize):
$movingAverages")
  }
}
```

Output:-

```
[info] running MovingAverage
Original Time Series: List(10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0)
Moving Average (window size = 3): List(20.0, 30.0, 40.0, 50.0, 60.0)
```

9. Write a program to compute frequency distribution and cumulative frequency of a dataset.

```
object FrequencyDistribution {
  def main(args: Array[String]): Unit = {
    // Sample dataset
    val data = List(5, 3, 6, 3, 5, 7, 3, 5, 6, 7, 8, 5, 6)

    // Step 1: Frequency Distribution
    val frequency =
data.groupBy(identity).mapValues(_.size).toSeq.sortBy(_._1)

    // Step 2: Cumulative Frequency
    val cumulativeFrequency = frequency.scanLeft((0, 0)) {
      case ((_, cumFreq), (value, freq)) => (value, cumFreq + freq)
```

```
    }.tail

    // Output
    println("Value\tFrequency\tCumulative Frequency")
    frequency.zip(cumulativeFrequency).foreach {
      case ((value, freq), (_, cumFreq)) =>
        println(s"$value\t$freq\t\t$cumFreq")
    }
  }
}
```

Output:-

```
Value   Frequency       Cumulative Frequency
3       3               3
5       4               7
6       3               10
7       2               12
8       1               13
```

10. Implement linear regression using Breeze. Fit a model to a small dataset and predict a value.

```
import breeze.linalg._
import breeze.numerics._

object LinearRegressionBreeze {
  def main(args: Array[String]): Unit = {
    // Feature vector (independent variable)
    val xData = DenseVector(1.0, 2.0, 3.0, 4.0, 5.0)

    // Target vector (dependent variable)
    val yData = DenseVector(2.0, 4.1, 6.0, 8.1, 10.1)

    // Add a column of ones to xData for the intercept term
    val ones = DenseVector.ones[Double](xData.length)
    val X = DenseMatrix.horzcat(ones.toDenseMatrix.t,
xData.toDenseMatrix.t)

    // Normal Equation: θ = (X^T X)^-1 X^T y
    val theta = inv(X.t * X) * X.t * yData
```

```
    println(f"Fitted model: y = ${theta(0)}%.4f + ${theta(1)}%.4f * x")

    // Predict y for a new x = 6
    val xNew = DenseVector(1.0, 6.0) // 1 for intercept, 6 as the new
feature value
    val yPred = xNew.t * theta

    println(f"Predicted value for x = 6: y = $yPred%.4f")
  }
}
```

Output:-

```
Fitted model: y = -0.0000 + 2.0200 * x
Predicted value for x = 6: y = 12.1200
```

11. Sort a dataset by a specific column and extract the top 5 rows.

```
object SortDatasetTop5 {
  def main(args: Array[String]): Unit = {

    // Sample dataset: List of rows (each row is a tuple or list)
    val data = List(
      List("A", 85, "X"),
      List("B", 90, "Y"),
      List("C", 75, "Z"),
      List("D", 88, "W"),
      List("E", 95, "V"),
      List("F", 80, "U"),
      List("G", 70, "T")
    )

    // Sort by the 2nd column (index 1) in descending order
    val sortedData = data.sortBy(row => -row(1).asInstanceOf[Int])

    // Extract top 5 rows
    val top5 = sortedData.take(5)

    // Print results
    println("Top 5 rows sorted by column 2 (descending):")
    top5.foreach(row => println(row.mkString("\t")))
  }
```

}

Output:-

```
Top 5 rows sorted by column 2 (descending):
E       95      V
B       90      Y
D       88      W
A       85      X
F       80      U
```

12. Perform logistic regression using Breeze. Classify a dataset with binary labels.

```
package example

import breeze.linalg._
import breeze.numerics._

object LogisticRegressionExample {
  def main(args: Array[String]): Unit = {
    // Features: 2D dataset (4 samples, 2 features)
    val Xraw = DenseMatrix((1.0, 2.0), (2.0, 1.0), (4.0, 5.0), (6.0, 7.0))
    val y = DenseVector(0.0, 0.0, 1.0, 1.0)

    // Add intercept term (column of ones)
    val ones = DenseMatrix.ones[Double](Xraw.rows, 1)
    val X = DenseMatrix.horzcat(ones, Xraw) // X becomes [1 x1 x2]

    // Initialize weights (theta)
    var theta = DenseVector.zeros[Double](X.cols)

    // Hyperparameters
    val learningRate = 0.1
    val iterations = 1000

    // Gradient Descent
    for (i <- 0 until iterations) {
      val z = X * theta
      val h = sigmoid(z)
      val error = h - y
      val gradient = X.t * error / y.length.toDouble
      theta -= learningRate * gradient
```

```
    }

    // Model output
    println(f"Final weights (theta): $theta")
    println()

    // Predict for new data
    val newSample = DenseVector(1.0, 3.0, 3.0) // includes bias term
    val predictionProb = sigmoid(theta.dot(newSample))
    val prediction = if (predictionProb >= 0.5) 1 else 0

    println(f"Predicted probability: $predictionProb%.4f")
    println(s"Predicted class: $prediction")
  }

  // Sigmoid function
  def sigmoid(z: DenseVector[Double]): DenseVector[Double] = {
    z.map(zi => 1.0 / (1.0 + math.exp(-zi)))
  }

  def sigmoid(z: Double): Double = 1.0 / (1.0 + math.exp(-z))
}
```

Output:-

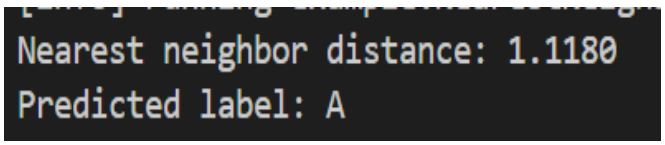
```
Final weights (theta): DenseVector(-5.538365290947838, 0.5221883903390814, 1.312735978626473)

Predicted probability: 0.4916
Predicted class: 0
```

13. Compute the Euclidean distance between two Breeze vectors. Use it for nearest neighbor classification.

```
package example

import breeze.linalg._
import breeze.numerics._

object NearestNeighbor {
  def main(args: Array[String]): Unit = {

    // Sample dataset: (features, label)
```

```
    val data = Seq(
      (DenseVector(1.0, 2.0), "A"),
      (DenseVector(2.0, 3.0), "A"),
      (DenseVector(4.0, 5.0), "B"),
      (DenseVector(5.0, 6.0), "B")
    )

    // New sample to classify
    val newSample = DenseVector(3.0, 3.5)

    // Compute Euclidean distances from newSample to all training points
    val distances = data.map { case (features, label) =>
      val distance = euclideanDistance(features, newSample)
      (distance, label)
    }

    // Find nearest neighbor
    val nearest = distances.minBy(_._1)
    val predictedLabel = nearest._2

    println(f"Nearest neighbor distance: ${nearest._1}%.4f")
    println(s"Predicted label: $predictedLabel")
  }

  // Function to compute Euclidean distance between two vectors
  def euclideanDistance(v1: DenseVector[Double], v2:
DenseVector[Double]): Double = {
    norm(v1 - v2)
  }
}
```

Output:-



14.Cluster a dataset into two groups using k-means clustering in Breeze.

```
package example

import breeze.linalg._
import breeze.numerics._
import scala.util.Random

object KMeansClustering {
  def main(args: Array[String]): Unit = {

    // Sample dataset: 2D points
    val data = Seq(
      DenseVector(1.0, 2.0),
      DenseVector(1.5, 1.8),
      DenseVector(5.0, 8.0),
      DenseVector(6.0, 9.0),
      DenseVector(1.2, 0.9),
      DenseVector(5.5, 8.5)
    )

    val k = 2  // Number of clusters
    val maxIterations = 100

    // Initialize cluster centers randomly
    val centroids = Random.shuffle(data).take(k).toArray

    var assignments = new Array[Int](data.length)
    var changed = true
    var iter = 0

    while (changed && iter < maxIterations) {
      changed = false
      iter += 1

      // Step 1: Assign each point to the nearest centroid
      for (i <- data.indices) {
        val distances = centroids.map(c => norm(data(i) - c))
```

```
        val nearest = distances.zipWithIndex.minBy(_._1)._2
        if (assignments(i) != nearest) {
          assignments(i) = nearest
          changed = true
        }
      }

      // Step 2: Recalculate centroids
      for (j <- 0 until k) {
        val assignedPoints = data.zip(assignments).collect { case (p, `j`)
=> p }
        if (assignedPoints.nonEmpty) {
          centroids(j) = assignedPoints.reduce(_ + _) /:/
assignedPoints.length.toDouble
        }
      }
    }

    // Print final cluster assignments
    println(s"\nK-Means clustering completed in $iter iterations.\n")
    data.zip(assignments).foreach { case (point, cluster) =>
      println(f"Point: $point%-20s  Cluster: $cluster")
    }
  }
}
```

Output:-

```
K-Means clustering completed in 2 iterations.

Point: DenseVector(1.0, 2.0)  Cluster: 1
Point: DenseVector(1.5, 1.8)  Cluster: 1
Point: DenseVector(5.0, 8.0)  Cluster: 0
Point: DenseVector(6.0, 9.0)  Cluster: 0
Point: DenseVector(1.2, 0.9)  Cluster: 1
Point: DenseVector(5.5, 8.5)  Cluster: 0
```

15. Create polynomial features from a dataset. Given a list of numbers (e.g., [1, 2, 3]), generate polynomial features up to degree 3 (e.g., [1, 1^2, 1^3, 2, 2^2, 2^3, 3, 3^2, 3^3]).

```
object PolynomialFeaturesExample {
  def main(args: Array[String]): Unit = {
    val numbers = List(1, 2, 3)
    val degree = 3

    val polyFeatures = numbers.flatMap { x =>
      (1 to degree).map(d => Math.pow(x, d))
    }

    println("Original numbers: " + numbers.mkString(", "))
    println("Polynomial features (degree 3): " +
polyFeatures.mkString(", "))
  }
}
```

Output:-

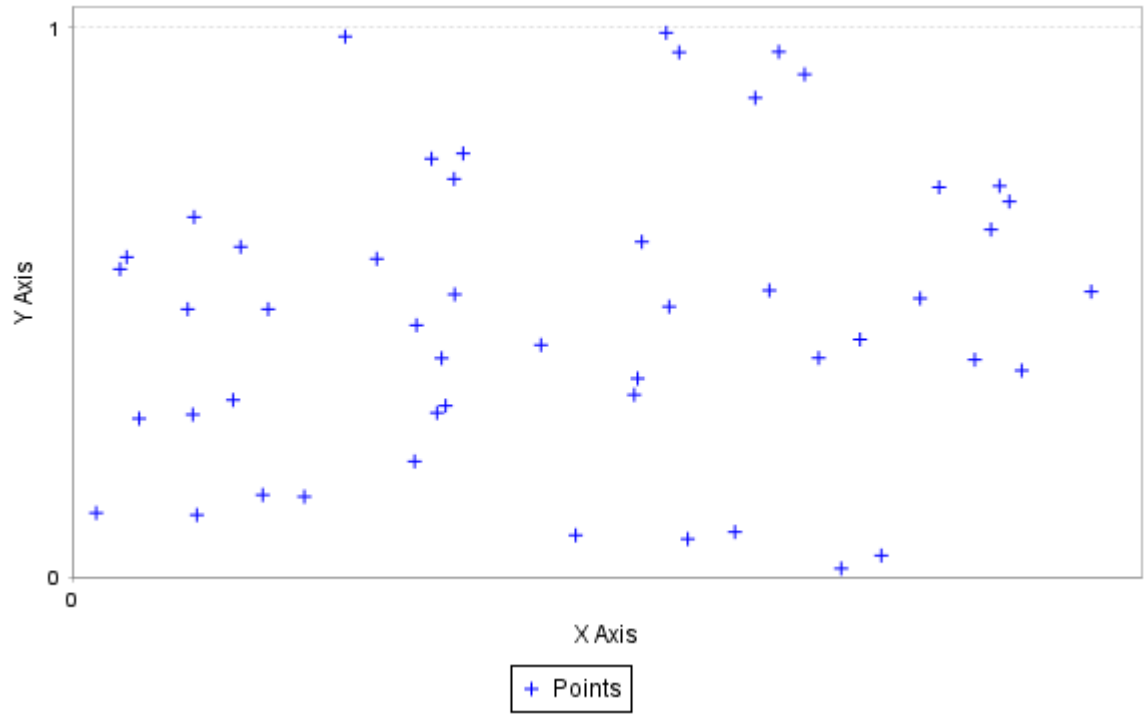
```
Original numbers: 1, 2, 3
Polynomial features (degree 3): 1.0, 1.0, 1.0, 2.0, 4.0, 8.0, 3.0, 9.0, 27.0
```

16. Create a scatter plot of random data using Breeze-viz. Label the axes and customize the color of points.

```
package example

import breeze.linalg._
import breeze.plot._
import scala.util.Random

object ScatterPlotExample {
  def main(args: Array[String]): Unit = {

    val f = Figure("Scatter Plot of Random Data")
    val p = f.subplot(0)

    val n = 50
```

```
    val x = DenseVector.rand[Double](n)
    val y = DenseVector.rand[Double](n)

    p += plot(x, y, '+', name = "Points", colorcode = "blue")
    p.xlabel = "X Axis"
    p.ylabel = "Y Axis"
    p.legend = true

    // Save plot to PNG file instead of displaying
    f.saveas("scatter_plot.png")  // Saves in project root

    println("✅ Scatter plot saved as 'scatter_plot.png'")
  }
}
```

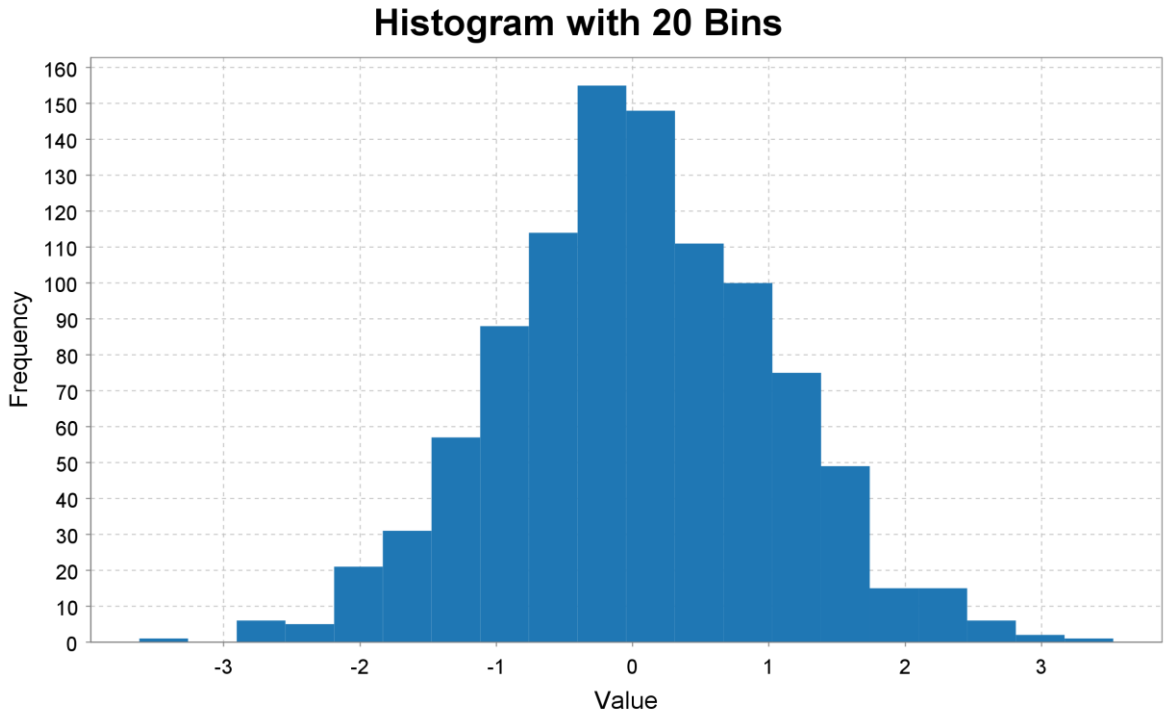
Output:-



17. Generate a histogram of a dataset using Breeze-viz. Experiment with different bin sizes.

```
package example

import breeze.linalg._
import breeze.plot._
import scala.util.Random

object HistogramExample {
  def main(args: Array[String]): Unit = {

    // Step 1: Generate a sample dataset (e.g., 1000 random numbers)
    val data = DenseVector.fill(1000)(Random.nextGaussian())

    // Step 2: Create a histogram plot
    val f = Figure("Histogram Example")
    val p = f.subplot(0)

    // Step 3: Plot the histogram with a specified number of bins
    val bins = 20 // Change this number to experiment (e.g., 10, 30, 50)
    p += hist(data, bins)

    // Step 4: Customize plot
    p.xlabel = "Value"
    p.ylabel = "Frequency"
    p.title = s"Histogram with $bins Bins"

    // Step 5: Save the figure as PNG
    f.saveas(s"histogram_bins_$bins.png", 300)

    // Optional: Display plot in a window
    f.refresh()
  }
}
```

Output:-

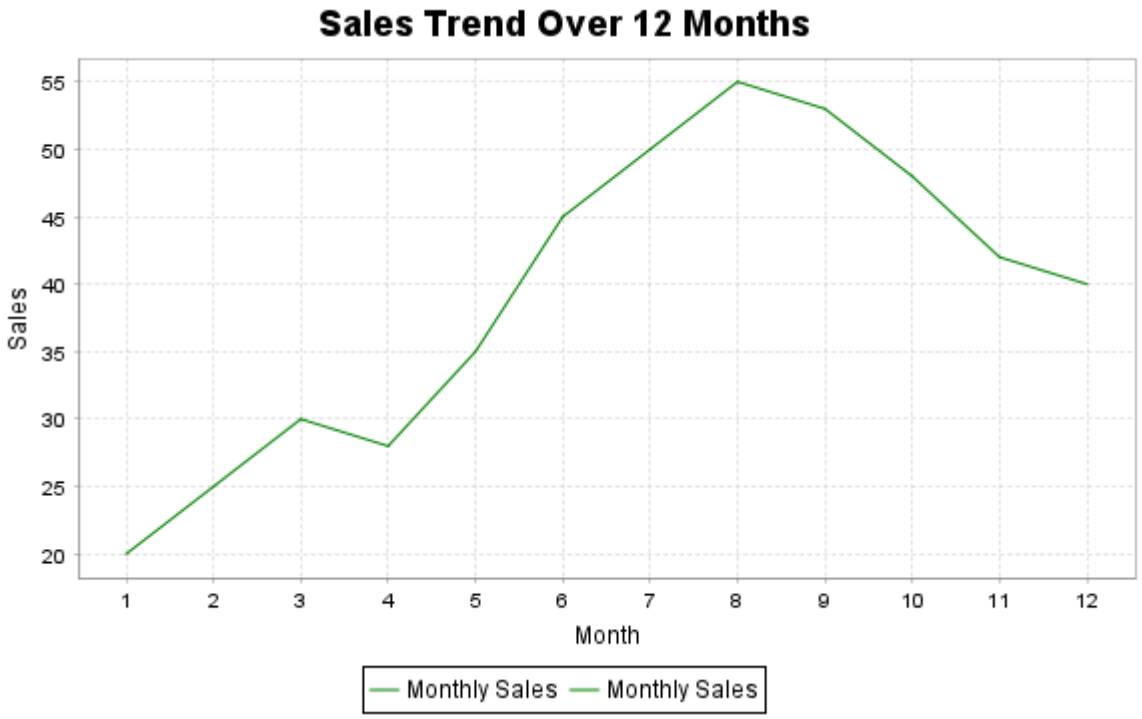


18. Plot a line graph for a dataset showing a trend over time.

```
package example

import breeze.linalg._
import breeze.plot._

object LineGraphExample extends App {
  // Sample time series data
  val months = DenseVector.range(1, 13).map(_.toDouble) // convert to
DenseVector[Double]
  val sales = DenseVector(20.0, 25.0, 30.0, 28.0, 35.0, 45.0, 50.0, 55.0, 53.0,
48.0, 42.0, 40.0)

  // Create plot
  val f = Figure()
  val p = f.subplot(0)
  p += plot(months, sales, name = "Monthly Sales", colorcode = "g")
  p.xlabel = "Month"
  p.ylabel = "Sales"
  p.title = "Sales Trend Over 12 Months"
```

```
  p.legend = true

  f.saveas("line_graph_sales.png")
}
```

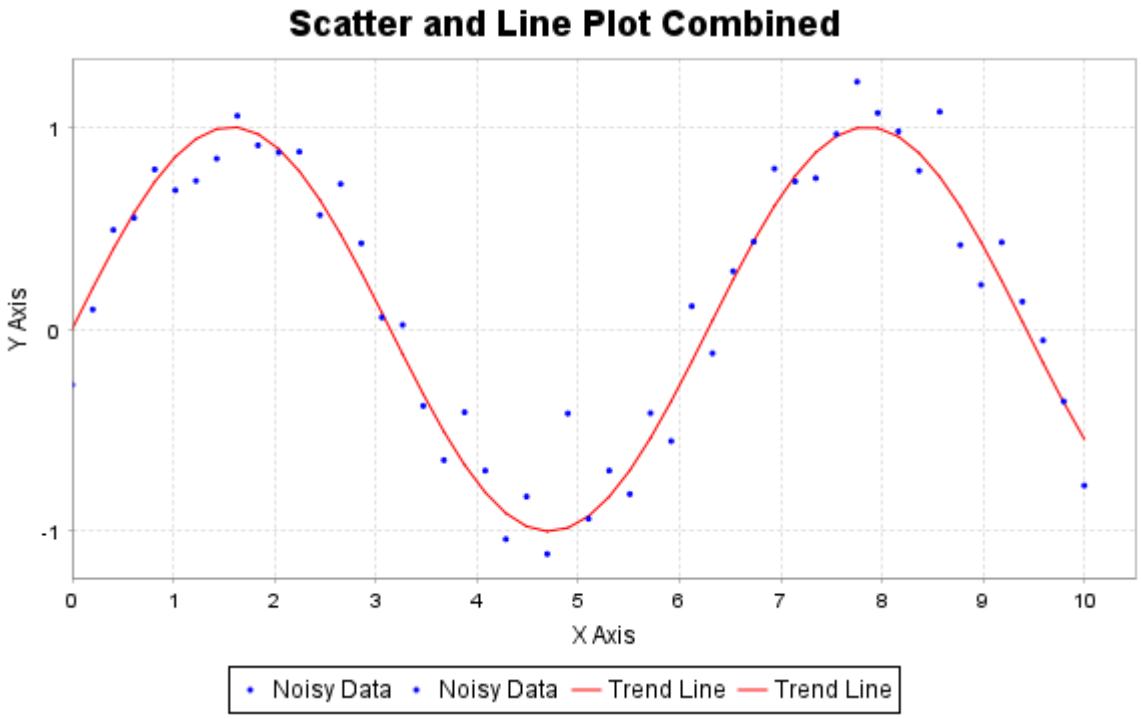
Output:-



19. Combine two plots (e.g., scatter and line plot) in a single visualization using Breeze-viz.

```
package example

import breeze.linalg._
import breeze.plot._

object CombinedPlotExample extends App {
  // Sample data
  val x = linspace(0.0, 10.0, 50)
  val y = x.map(xi => math.sin(xi) + scala.util.Random.nextGaussian() * 0.2) //
noisy sine wave

  // Create the figure and plot
  val f = Figure("Combined Plot")
  val p = f.subplot(0)

  // Scatter plot: blue points
  p += plot(x, y, '.', name = "Noisy Data", colorcode = "b")

  // Line plot: smooth sine curve
  val ySmooth = x.map(math.sin)
  p += plot(x, ySmooth, name = "Trend Line", colorcode = "r")

  // Titles and labels
  p.title = "Scatter and Line Plot Combined"
  p.xlabel = "X Axis"
  p.ylabel = "Y Axis"
  p.legend = true

  // Save as image (optional)
  f.saveas("combined_plot.png")
}
```

Output:-


Scatter and Line Plot Combined
Y Axis
X Axis
1
0
-1
0
1
2
3
4
5
6
7
8
9
10
Noisy Data
Noisy Data
Trend Line
Trend Line

20. Find the correlation between two lists of numbers. Implement the formula for Pearson correlation coefficient.

```
object PearsonCorrelation {
  def main(args: Array[String]): Unit = {
    val x = List(1.0, 2.0, 3.0, 4.0, 5.0)
    val y = List(2.0, 4.0, 6.0, 8.0, 10.0)

    val correlation = pearsonCorrelation(x, y)
    println(f"Pearson correlation coefficient: $correlation%.4f")
  }

  def pearsonCorrelation(x: List[Double], y: List[Double]): Double = {
    require(x.length == y.length, "Lists must be the same length")

    val n = x.length
    val meanX = x.sum / n
    val meanY = y.sum / n

    val numerator = x.zip(y).map { case (xi, yi) => (xi - meanX) * (yi - meanY)
}.sum
    val denominatorX = math.sqrt(x.map(xi => math.pow(xi - meanX, 2)).sum)
    val denominatorY = math.sqrt(y.map(yi => math.pow(yi - meanY, 2)).sum)

    numerator / (denominatorX * denominatorY)
  }
}
```

Output:-

```
Pearson correlation coefficient: 1.0000
```