Scala for Data Science - Complete Practical Guide

Table of Contents

Project Setup

build.sbt Configuration

```scala
ThisBuild / version := "0.1.0-SNAPSHOT"
ThisBuild / scalaVersion := "2.13.10"

lazy val root = (project in file("."))
  .settings(
    name := "ScalaDataScience",
    libraryDependencies ++= Seq(
      "org.scalanlp" %% "breeze" % "2.1.0",
      "org.scalanlp" %% "breeze-viz" % "2.1.0",
      "com.github.tototoshi" %% "scala-csv" % "1.3.10",
      "org.apache.commons" % "commons-math3" % "3.6.1"
    )
  )
```

Required Imports

```scala
import breeze.linalg._
import breeze.numerics._
import breeze.stats._
import breeze.plot._
import scala.util.Random
import scala.io.Source
import com.github.tototoshi.csv._
import java.io.{File, PrintWriter}
```

Module 0-1: Welcome and Setup

Objective

Set up Scala and SBT, create a welcome message for data scientists.

```scala
def welcomeMessage(): Unit = {
  println("=" * 50)
  println("Welcome to Scala for Data Science!")
  println("Empowering Data Scientists with Functional Programming")
  println("=" * 50)
}
```

Key Learning Points:

- Basic Scala syntax and string manipulation
- Using repetition operators for formatting
- Introduction to functional programming concepts

Module 2: Basic Statistical Calculations

Objective

Calculate mean, median, and mode using Scala collections.

```scala
def calculateBasicStats(numbers: List[Double]): Map[String, Double] = {
  val sorted = numbers.sorted
  val n = numbers.length

  val mean = numbers.sum / n

  val median = if (n % 2 == 0) {
    (sorted(n/2 - 1) + sorted(n/2)) / 2.0
  } else {
    sorted(n/2)
  }

  val mode = numbers.groupBy(identity)
    .maxBy(_._2.length)._1

  Map("mean" -> mean, "median" -> median, "mode" -> mode)
}
```

Key Learning Points:

- Working with Scala collections (List, Map)
- Conditional expressions with if-else
- Higher-order functions like `groupBy`, `maxBy`
- Pattern matching and functional transformations

Example Usage:

```scala
val numbers = List(1.0, 2.0, 3.0, 4.0, 5.0, 3.0, 2.0)
val stats = calculateBasicStats(numbers)
// Output: Map(mean -> 2.857, median -> 3.0, mode -> 2.0)
```

Module 3: Variance and Standard Deviation

Objective

Generate random datasets and calculate variance and standard deviation.

```scala
def generateRandomDataset(size: Int = 10): List[Double] = {
  val random = new Random()
  (1 to size).map(_ => random.nextDouble() * 100).toList
}

def calculateVarianceAndStdDev(numbers: List[Double]): (Double, Double) = {
  val mean = numbers.sum / numbers.length
  val variance = numbers.map(x => math.pow(x - mean, 2)).sum / numbers.length
  val stdDev = math.sqrt(variance)
  (variance, stdDev)
}
```

Key Learning Points:

- Random number generation in Scala
- Mathematical operations and transformations
- Tuple return types
- Map operations on collections

Example Usage:

```scala
val randomData = generateRandomDataset(10)
val (variance, stdDev) = calculateVarianceAndStdDev(randomData)
println(s"Variance: $variance, Standard Deviation: $stdDev")
```

Module 4: Dense Vector Operations

Objective

Create dense vectors using Breeze and perform basic operations.

```scala
def vectorOperations(): Unit = {
  val vector1 = DenseVector(1.0, 2.0, 3.0, 4.0, 5.0)
  val vector2 = DenseVector(2.0, 3.0, 4.0, 5.0, 6.0)

  val sum = breeze.linalg.sum(vector1)
  val mean = breeze.stats.mean(vector1)
  val dotProduct = vector1 dot vector2

  println(s"Vector 1: $vector1")
  println(s"Vector 2: $vector2")
  println(s"Sum: $sum")
  println(s"Mean: $mean")
  println(s"Dot Product: $dotProduct")
}
```

Key Learning Points:

- Introduction to Breeze library
- Dense vector creation and manipulation
- Vector operations: sum, mean, dot product
- Linear algebra fundamentals

Module 5: Matrix Operations

Objective

Generate random matrices and compute transpose and determinant.

scala

```scala
def matrixOperations(): Unit = {
  val random = new Random()
  val matrix = DenseMatrix.rand(3, 3)

  val transpose = matrix.t
  val determinant = det(matrix)

  println(s"Original Matrix:\n$matrix")
  println(s"Transpose:\n$transpose")
  println(s"Determinant: $determinant")
}
```

Key Learning Points:

- Matrix creation and random generation
- Matrix transpose operation
- Determinant calculation
- Matrix display and formatting

Module 6: Matrix Slicing

Objective

Extract sub-matrices and calculate row and column sums.

scala

```scala
def matrixSlicingOperations(): Unit = {
  val matrix = DenseMatrix((1.0, 2.0, 3.0, 4.0),
                           (5.0, 6.0, 7.0, 8.0),
                           (9.0, 10.0, 11.0, 12.0))

  // Extract sub-matrix (first 2 rows, first 3 columns)
  val subMatrix = matrix(0 to 1, 0 to 2)

  // Calculate row sums and column sums
  val rowSums = sum(matrix(*, ::))
  val colSums = sum(matrix(::, *))

  println(s"Original Matrix:\n$matrix")
  println(s"Sub-matrix:\n$subMatrix")
  println(s"Row sums: $rowSums")
  println(s"Column sums: $colSums")
}
```

Key Learning Points:

- Matrix indexing and slicing syntax
- Range operations (0 to 1, 0 to 2)
- Row and column aggregation operations
- Wildcard operators (* and ::)

Module 7: Element-wise Matrix Operations

Objective

Perform element-wise addition, subtraction, multiplication, and division.

scala

```scala
def elementWiseMatrixOperations(): Unit = {
  val matrix1 = DenseMatrix((1.0, 2.0), (3.0, 4.0))
  val matrix2 = DenseMatrix((5.0, 6.0), (7.0, 8.0))

  val addition = matrix1 + matrix2
  val subtraction = matrix1 - matrix2
  val multiplication = matrix1 *:* matrix2  // element-wise multiplication
  val division = matrix1 /:/ matrix2        // element-wise division

  println(s"Matrix 1:\n$matrix1")
  println(s"Matrix 2:\n$matrix2")
  println(s"Addition:\n$addition")
  println(s"Subtraction:\n$subtraction")
  println(s"Element-wise Multiplication:\n$multiplication")
  println(s"Element-wise Division:\n$division")
}
```

Key Learning Points:

- Element-wise vs. matrix multiplication
- Breeze operator syntax (:, /:/)
- Matrix arithmetic operations
- Broadcasting concepts

Module 8: CSV File Operations

Objective

Read CSV files and calculate statistics for numeric columns.

scala

```scala
def createSampleCSV(): Unit = {
  val data = List(
    List("Name", "Age", "Salary", "Experience"),
    List("Alice", "25", "50000", "2"),
    List("Bob", "30", "75000", "5"),
    List("Charlie", "35", "90000", "8"),
    List("Diana", "28", "65000", "4"),
    List("Eve", "32", "80000", "6")
  )

  val writer = CSVWriter.open(new File("sample_data.csv"))
  writer.writeAll(data)
  writer.close()
}

def readCSVAndCalculateStats(): Unit = {
  createSampleCSV() // Create sample data first

  val reader = CSVReader.open(new File("sample_data.csv"))
  val data = reader.all()
  reader.close()

  val headers = data.head
  val rows = data.tail

  // Find numeric columns
  val numericColumns = List("Age", "Salary", "Experience")

  numericColumns.foreach { colName =>
    val colIndex = headers.indexOf(colName)
    if (colIndex != -1) {
      val values = rows.map(_(colIndex).toDouble)
      val stats = calculateBasicStats(values)
      println(s"Statistics for $colName:")
      stats.foreach { case (stat, value) => println(s"  $stat: $value") }
      println()
    }
  }
}
```

Key Learning Points:

- File I/O operations in Scala
- CSV reading and writing with scala-csv library
- Data type conversion (String to Double)
- Column-wise data processing

Module 9: Handle Missing Values

Objective

Handle missing values by replacing them with column means.

scala

```scala
def handleMissingValues(): Unit = {
  // Simulate data with missing values (represented as NaN)
  val dataWithMissing = List(
    List(1.0, 2.0, Double.NaN, 4.0),
    List(5.0, Double.NaN, 7.0, 8.0),
    List(9.0, 10.0, 11.0, Double.NaN),
    List(13.0, 14.0, 15.0, 16.0)
  )

  println("Original data with missing values:")
  dataWithMissing.foreach(row => println(row.map(x => if (x.isNaN) "NaN" else x.toString).mkStr

  // Replace missing values with column mean
  val cleanedData = (0 until dataWithMissing.head.length).map { colIndex =>
    val column = dataWithMissing.map(_(colIndex))
    val validValues = column.filter(!_.isNaN)
    val mean = if (validValues.nonEmpty) validValues.sum / validValues.length else 0.0

    column.map(value => if (value.isNaN) mean else value)
  }.transpose

  println("\nData after replacing missing values with column mean:")
  cleanedData.foreach(row => println(row.mkString(", ")))
}
```

Key Learning Points:

- Handling NaN values in Scala
- Column-wise data imputation strategies

- Data transformation and cleaning techniques
- Transpose operations on nested lists

Module 10: Filter Data by Threshold

Objective

Filter dataset rows based on column value thresholds.

```scala
def filterRowsByThreshold(): Unit = {
  val data = List(
    ("Alice", 25, 50000),
    ("Bob", 30, 75000),
    ("Charlie", 35, 90000),
    ("Diana", 28, 65000),
    ("Eve", 32, 80000)
  )

  val salaryThreshold = 70000
  val filteredData = data.filter(_._3 > salaryThreshold)

  println(s"Original data: ${data.length} records")
  println(s"Filtered data (salary > $salaryThreshold): ${filteredData.length} records")
  filteredData.foreach(println)
}
```

Key Learning Points:

- Tuple access and manipulation
- Filtering operations with predicates
- Data querying and selection
- Lambda expressions and anonymous functions

Module 11: Text Processing

Objective

Tokenize text and count word frequency.

scala

```scala
def createSampleTextFile(): Unit = {
  val sampleText = """
    Scala is a modern multi-paradigm programming language designed to express common programmir
    Scala smoothly integrates features of object-oriented and functional languages.
    Data science with Scala provides powerful tools for data manipulation and analysis.
    Scala's functional programming features make it ideal for data processing tasks.
  """

  val writer = new PrintWriter(new File("sample_text.txt"))
  writer.write(sampleText)
  writer.close()
}

def tokenizeAndCountWords(): Unit = {
  createSampleTextFile()

  val source = Source.fromFile("sample_text.txt")
  val text = source.mkString
  source.close()

  val words = text.toLowerCase
    .replaceAll("[^a-zA-Z\\s]", "")
    .split("\\s+")
    .filter(_.nonEmpty)

  val wordCount = words.groupBy(identity).mapValues(_.length)
  val sortedWordCount = wordCount.toSeq.sortBy(-_._2)

  println("Word frequency (top 10):")
  sortedWordCount.take(10).foreach { case (word, count) =>
    println(s"$word: $count")
  }
}
```

Key Learning Points:

- File reading and text processing
- Regular expressions for text cleaning
- String manipulation methods
- Frequency counting and sorting algorithms

Module 12: One-hot Encoding

Objective

Implement one-hot encoding for categorical data.

```scala
def oneHotEncoding(): Unit = {
  val categories = List("Red", "Green", "Blue", "Red", "Blue", "Green", "Red")
  val uniqueCategories = categories.distinct.sorted

  println(s"Original categories: $categories")
  println(s"Unique categories: $uniqueCategories")

  val oneHotEncoded = categories.map { category =>
    uniqueCategories.map(unique => if (unique == category) 1 else 0)
  }

  println("One-hot encoded:")
  println(uniqueCategories.mkString("\t"))
  oneHotEncoded.foreach(encoding => println(encoding.mkString("\t")))
}
```

Key Learning Points:

- Categorical data transformation
- One-hot encoding implementation
- Binary representation of categories
- Data preprocessing for machine learning

Module 13: Scatter Plot

Objective

Create scatter plots with custom styling using Breeze-viz.

scala

```scala
def createScatterPlot(): Unit = {
  val random = new Random()
  val x = DenseVector.rand(50) * 10
  val y = x + DenseVector.rand(50) * 3

  val f = Figure()
  val p = f.subplot(0)
  p += plot(x, y, '.')
  p.xlabel = "X Values"
  p.ylabel = "Y Values"
  p.title = "Scatter Plot with Custom Colors"

  println("Scatter plot created! Check the visualization window.")
}
```

Key Learning Points:

- Data visualization with Breeze-viz
- Scatter plot creation and styling
- Figure and subplot management
- Axis labeling and titles

Module 14: Histogram

Objective

Generate histograms with different bin sizes.

scala

```scala
def createHistogram(): Unit = {
  val random = new Random()
  val data = DenseVector.rand(1000).map(_ * 100)

  val f = Figure()
  val p = f.subplot(0)
  p += hist(data, bins = 20)
  p.xlabel = "Values"
  p.ylabel = "Frequency"
  p.title = "Histogram with 20 bins"

  println("Histogram created! Experiment with different bin sizes by changing the 'bins' parame
}
```

Key Learning Points:

- Distribution visualization
- Histogram creation and customization
- Bin size selection strategies
- Frequency analysis

Module 15: Line Plot

Objective

Create line plots for time series data.

scala

```scala
def createLinePlot(): Unit = {
  val time = linspace(0, 10, 100)
  val values = time.map(t => math.sin(t) + math.random() * 0.3)

  val f = Figure()
  val p = f.subplot(0)
  p += plot(time, values)
  p.xlabel = "Time"
  p.ylabel = "Values"
  p.title = "Time Series Line Plot"

  println("Line plot created! Shows a sine wave with noise over time.")
}
```

Key Learning Points:

- Time series visualization
- Line plot creation
- Mathematical functions in plotting
- Noise addition to data

Module 16: Combined Plots

Objective

Combine multiple plot types in a single visualization.

scala

```scala
def createCombinedPlots(): Unit = {
  val x = linspace(0, 10, 50)
  val y1 = x.map(math.sin)
  val y2 = x.map(t => math.cos(t) + scala.util.Random.nextGaussian() * 0.1)

  val f = Figure()
  val p = f.subplot(0)

  // Line plot
  p += plot(x, y1, name = "sin(x)")

  // Scatter plot
  p += plot(x, y2, '.', name = "cos(x) + noise")

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

  println("Combined plot created! Shows both line and scatter plots together.")
}
```

Key Learning Points:

- Multiple plot overlay techniques
- Legend creation and management
- Plot naming and identification
- Complex visualization composition

Utility Functions

Additional Helper Functions

scala

```scala
object DataUtils {

  // Helper function to generate synthetic datasets
  def generateSyntheticDataset(rows: Int, cols: Int): DenseMatrix[Double] = {
    DenseMatrix.rand(rows, cols) * 100
  }

  // Helper function for data normalization
  def normalizeColumn(data: DenseVector[Double]): DenseVector[Double] = {
    val mean = breeze.stats.mean(data)
    val std = breeze.stats.stddev(data)
    (data - mean) / std
  }

  // Helper function to calculate correlation coefficient
  def correlationCoefficient(x: DenseVector[Double], y: DenseVector[Double]): Double = {
    val n = x.length
    val sumX = sum(x)
    val sumY = sum(y)
    val sumXY = sum(x *:* y)
    val sumX2 = sum(x *:* x)
    val sumY2 = sum(y *:* y)

    val numerator = n * sumXY - sumX * sumY
    val denominator = math.sqrt((n * sumX2 - sumX * sumX) * (n * sumY2 - sumY * sumY))

    numerator / denominator
  }
}

// Case class for structured data handling
case class DataPoint(id: Int, features: DenseVector[Double], label: String)

object DataPoint {
  def fromCSVRow(row: List[String], featureIndices: List[Int], labelIndex: Int): DataPoint = {
    val id = row(0).toInt
    val features = DenseVector(featureIndices.map(i => row(i).toDouble).toArray)
    val label = row(labelIndex)
    DataPoint(id, features, label)
  }
}
```

Complete Main Function

```scala
def main(args: Array[String]): Unit = {
  println("Starting Scala Data Science Examples...\n")

  // Module 0 & 1
  welcomeMessage()

  // Module 2
  println("\n--- Module 2: Basic Statistics ---")
  val numbers = List(1.0, 2.0, 3.0, 4.0, 5.0, 3.0, 2.0)
  val stats = calculateBasicStats(numbers)
  println(s"Numbers: $numbers")
  stats.foreach { case (stat, value) => println(s"$stat: $value") }

  // Module 3
  println("\n--- Module 3: Variance and Standard Deviation ---")
  val randomData = generateRandomDataset(10)
  val (variance, stdDev) = calculateVarianceAndStdDev(randomData)
  println(s"Random dataset: $randomData")
  println(s"Variance: $variance")
  println(s"Standard deviation: $stdDev")

  // Continue with all other modules...
  // [Additional module calls here]

  println("\nAll modules completed successfully!")
}
```

How to Convert to PDF

Method 1: Using Pandoc (Recommended)

1. **Install Pandoc**: Download from pandoc.org
2. **Save this content** as `scala-ds-guide.md`
3. **Convert to PDF**:

```bash
pandoc scala-ds-guide.md -o scala-ds-guide.pdf --pdf-engine=xelatex
```

Method 2: Using Online Converters

1. **Copy the markdown content** from this document
2. **Visit an online converter** like:
   - Markdown to PDF
   - Pandoc Try
   - Dillinger.io (export as PDF)

Method 3: Using VS Code

1. **Install the "Markdown PDF" extension** in VS Code
2. **Save this content** as a `.md` file
3. **Right-click** and select "Markdown PDF: Export (pdf)"

Method 4: Using GitHub

1. **Create a new repository** on GitHub
2. **Upload this content** as `README.md`
3. **Use GitHub's export features** or online tools that can convert GitHub README to PDF

Execution Instructions

1. **Create a new SBT project**:

   bash

   ```
   mkdir scala-data-science
   cd scala-data-science
   sbt new scala/scala-seed.g8
   ```

2. **Replace the** `build.sbt` with the configuration provided above
3. **Create the main Scala file** in `src/main/scala/ScalaDataScience.scala`
4. **Run the project**:

   bash

   ```
   sbt compile
   sbt run
   ```

5. **For visualization modules**, ensure you have a display environment or run on a system with GUI support.

Additional Resources

- **Breeze Documentation**: github.com/scalanlp/breeze
- **Scala Documentation**: docs.scala-lang.org
- **SBT Documentation**: www.scala-sbt.org
- **Scala for Data Science**: Various online tutorials and books

---

This guide provides a comprehensive introduction to using Scala for data science tasks, covering fundamental operations, data manipulation, statistical analysis, and visualization techniques.