Дата последней сборки документа:
13 декабря 2020 г. в 20:09

Заметки. Практика использования и наиболее полезные конструкции языка Scala

Подвойский А.О.

Здесь приводятся заметки по некоторым вопросам, касающимся машинного обучения, анализа данных, программирования на языках Scala и прочим сопряженным вопросам так или иначе, затрагивающим работу с данными.

Содержание

1. Установка SDK

SDKMAN (Software Development Kit Manager) https://sdkman.io/ – Очень полезная утилита для работы scala-средой.

```
curl -s "https://get.sdkman.io" | bash
source "$HOME/.sdkman/bin/sdkman-init.sh"
sdk version
```

2. Установка библиотек для Scala

Например, библиотеку `breeze` https://github.com/scalanlp/breeze/wiki/Installation, близкую по своему функционалу к библиотеке `numpy` языка `Python`, можно установить с помощью `sbt` следующим образом

build.sbt

```
name := "project_name"
version := "0.1"
scalaVersion := "2.13.3"

libraryDependencies ++= Seq(
    // Last stable release
    "org.scalanlp" %% "breeze" % "1.1",
    // Native libraries are not included by default. add this if you want them
    // Native libraries greatly improve performance, but increase jar sizes.
    // It also packages various blas implementations, which have licenses that may or may not
    // be compatible with the Apache License. No GPL code, as best I know.
    "org.scalanlp" %% "breeze-natives" % "1.1",
    // The visualization library is distributed separately as well.
    // It depends on LGPL code
    "org.scalanlp" %% "breeze-viz" % "1.1"
)
```

Если используется IDE IntelliJ IDEA, то файл `build.sbt` должен располагаться в директории проекта, например, `C:‣Users‣ADM‣IdeaProjects‣project_name`. Тогда после запуска сессии IDEA будут доступны все библиотеки.

Теперь можно запустить сессию в директории с файлом `build.sbt` командной

```
$ sbt console
scala> import breeze.linalg._
scala> val v = DenseVector(1.0, 2.0, 3.0)
```

К слову, есть полезная шпаргалка по `breeze` https://github.com/scalanlp/breeze/wiki/Linear-Algebra-Cheat-Sheet

3. Базовый sbt-файл

Зависимости проекта описываются в файле build.sbt

build.sbt

```
scalaVersion := "2.13.3"

libraryDependencies ++= Seq(
  "org.scalanlp" %% "breeze" % "1.1",
  "org.scalanlp" %% "breeze-natives" % "1.1",
  "org.scalanlp" %% "breeze-viz" % "1.1",
  "org.plotly-scala" %% "plotly-render" % "0.8.0",
  //"org.apache.spark" %% "spark-core" % "2.2.3" % "provided",
  //"org.apache.spark" %% "spark-sql" % "2.2.3" % "provided",
  //"org.vegas-viz" %% "vegas" % "0.3.11"
)
```

При работе со Scala-проектом с помощью `sbt` или IntelliJ IDEA версия языка определяется параметром `scalaVersion` в файле сборки `build.sbt`, например

```
scalaVersion := "2.12.12"
...
```

Остается только при запуске сессии в REPL набрать `sbt console` (а не `scala`), чтобы загрузить указанную версию Scala и все зависимости проекта.

То есть для того чтобы использовать разные версии Scala в разных проектах нужно указать нужную версию Scala в файле сборки `build.sbt`.

К слову, узнать версию Scala в сессии можно так

```
scala.util.Properties.versionNumberString  // String = 2.12.12
scala.util.Properties.versionMsg  // String = Scala library version 2.12.12 -- Copyright
    2002-2020, LAMP/EPFL and Lightbend, Inc.
```

На MacOS для того чтобы выяснить доступные версии Scala можно воспользоваться менеджером пакета `brew`

```
brew search scala
```

4. Компиляция программ на Scala

Пусть есть программа такая программа

```
object Hello extends App {
  println("Hello, world")
}
```

Скомпилировать эту программу можно с помощью утилиты командной строки `scalac`

```
scalac Hello.scala
```

Затем можно запустить программу с помощью утилиты командной строки `scala` °

```
scala Hello
```

После этого в рабочей директории появятся файлы с расширениями `Hello.class`, `'Hello$.class'`, `'Hello$delayedInit$body.class'`

5. Сборка jar-файлов в Spark-проектах

Создадим простое Spark-приложение

SimpleApp.scala

```
import org.apache.spark.sql.SparkSession

object SimpleApp {
  def main(args: Array[String]) = {
    val logFile = "./req.txt"
    // создаем Spark-сессию
    val spark = SparkSession.builder.appName("Simple Application").getOrCreate()
    val logData = spark.read.textFile(logFile).cache()
    val numsA = logData.filter(line => line.contains("a")).count()
    val numsB = logData.filter(line => line.contains("b")).count()
    println(s"--> Lines with a: $numsA, Lines with b: $numsB")
    spark.stop()
  }
}
```

В файл сборки `build.sbt` следует добавить следующие строки

Пример файла build.sbt

```
name := "SparkML"
```

```
version := "1.0"

scalaVersion := "2.12.12"

libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-sql" % "3.0.1" % "provided",
  "org.apache.spark" %% "spark-mllib_2.12" % "3.0.1" % "provided"
)
```

Для того чтобы `sbt` работал корректно, требуется разместить `SimpleApp.scala` и `build.sbt` следующим образом:

- файл `build.sbt` должен лежать в корне проекта,
- а scala-скрипт – по пути `src▸main▸scala▸SimpleApp.scala`.

Теперь можно упаковать приложение

```
sbt package
```

Для запуска scala-приложения используется `spark-submit`

```
spark-submit \
  --class "SimpleApp" \
  --master local \
  target/scala-2.12/simple-project_2.12-1.0.jar
```

Для запуска python-сценария следует набрать

```
spark-submit \
  --master local \
  SimpleApp.py
```

6. Работа с языком Scala в JupyterLab

Для того, чтобы JupyterLab поддерживал код на Scala требуется установить ядро `spylon-kernel`

```
# Step 1: Install spylon kernel
pip install spylon-kernel

# Step 2: create a kernel spec
python -m spylon_kernel install

# Step 3: start jupyter notebook
jupyter notebook
```

Посмотреть установленные ядра можно так

```
jupyter kernelspec list
```

В некотрых случаях удобнее работать с almond https://almond.sh/docs/try-docker – это Scala-ядро для Jupyter. Проще всего воспользоваться docker-образом

```
docker run -it --rm -p 8888:8888 almondsh/almond:latest
```

Можно указать конкретную версию almond или Scala

```
docker run -it --rm -p 8888:8888 almondsh/almond:0.10.9
docker run -it --rm -p 8888:8888 almondsh/almond:0.10.9-scala-2.12.8
```

Затем нужно будет открыть в браузере вкладку с адресом, который будет указан в логах. Для примера начнем работу с библиотекой `plotly` https://github.com/alexarchambault/plotly-scala

в сеансе Jupyter

```
import $ivy.`org.plotly-scala::plotly-almond:0.8.0` // <-- NB: динамическое подключение библиотеки
import plotly._
import plotly.element._
import plotly.layout._
import plotly.Almond._

val (x, y) = Seq(
  "Banana" -> 10,
  "Apple" -> 8,
  "Grapefruit" -> 5
).unzip

Bar(x, y).plot()
```

Узнать домашнюю директорию Spark можно так

```
echo 'sc.getConf.get("spark.home")' | spark-shell
```

Для поддержки Spark в almond необходимо добавить следующие строки (ВАЖНО: ядро и кластер Spark должны использовать одну и ту же версию Scala) https://github.com/almond-sh/almond/blob/master/docs/pages/usage-spark.md

в сеансе Jupyter. Для поддержки Spark

```
import $ivy.`org.apache.spark::spark-sql:2.4.0` // Or use any other 2.x version here
import $ivy.`sh.almond::almond-spark:@VERSION@` // Not required since almond 0.7.0 (will be automatically added when importing spark)
```

В самом простом случае можно воспользоваться web-интерфейсом `binder` https://mybinder.org/, доступного на странице проекта almond (по состоянию на 02.12.20 Spark совместим только с версией Scala 2.11).

Подключить `breeze` в сеансе JupyterLab на базе binder можно такой конструкцией

```
import $ivy.`org.scalanlp::breeze:1.0`

import breeze.linalg.{DenseMatrix, DenseVector}
import breeze.stats.regression.{leastSquares, lasso}
...
```

7. Работа со Scala через Ammonite

`Ammonite` https://ammonite.io/ на высоком уровне абстракции (очень грубо) представляет собой командную REPL-оболочку для Scala (на самом деле возможности гораздо шире).

8. Приемы использования библиотеки Breeze

Быстрое введение в библиотеку `Breeze` можно найти здесь https://github.com/scalanlp/breeze/wiki/Quickstart, а шпаргалку по работе с инструментами линейной алгебры по адресу https://github.com/scalanlp/breeze/wiki/Linear-Algebra-Cheat-Sheet.

Чтобы создать полносвязанный вектор или матрицу можно воспользоваться следующими приемами

```
import breeze.linalg._

DenseVector.ones[Double](5)
// np.ones(5) <-- numpy

DenseVector.fill(3){5}  // или просто DenseVector.fill(3)(5)
// np.ones(3)*5 <-- numpy

DenseVector.fill(3){scala.math.sin(10)}
// np.ones(3)*np.sin(10)

DenseMatrix.ones[Double](3,2)
// np.ones((3,2)) <-- numpy

DenseMatrix((1.0, 2.0), (3.0, 4.0))
// np.array([[1.0, 2.0], [3.0, 4.0]]) <-- numpy
```

Для диапазона

```
linspace(1,5,10)
// np.linspace(1,5,10 <-- numpy)
```

Транспонирование векторов и матриц

```
DenseVector(1.to(5): _*).t
// np.array(range(1,6)).reshape(-1, 1)

DenseMatrix((10, 20, 30), (40, 50, 60)).t
// np.array([[10, 20, 30], [40, 50, 60]]).T
```

Можно использовать приемы генерации значений таблицы на лету

```
DenseMatrix.tabulate(3, 2){ case (i, j) => i + j}
```

На ванильном Python генерацию на лету можно было бы реализовать так

```
def tabulate(n: int, m: int) -> np.array:
    arr = []
    for i in range(n):
        row = []
        for j in range(m):
            row.append(i + j)
        arr.append(row)
    return np.array(arr)
```

Создать матрицу на базе массива можно так

```
val mtx = new DenseMatrix(2, 3, Array[Int](10, 20, 30, 40, 50, 60)) // обязательно new!
```

Для создания векторов и матриц, заполненных равномерно распределенными псевдослучайными числами используется метод `rand`

```
DenseVector.rand(3)
// np.random.rand(3)
// или
DenseMatrix.rand(3, 2)
// np.random.rand(3, 2)
```

Чтение и запись векторов и матриц

```
val mtx = DenseMatrix.rand(3, 2)
mtx(1, 1) // 1-ая строка и 1-ый столбец

val v = DenseVector.rand(10)
v.slice(1,5) // или a(1 to 4) или a(1 until 5)
// v[1:5] <-- питру; правая граница не включается!

v(5 to 0 by -1)
// v[5::-1]

v(1 to -1) // v[1:]

v(-1) // v[-1] последний элемент

v(::, 2) // v[:, 2]
```

Примеры других манипуляций

```
mtx.reshape(3, 2) // как и в питру

// разворачивание матрицы в вектор
mtx.toDenseVector // mtx.flatten()

// копирование нижнего треугольника данных
lowerTriangular(mtx) // np.tril(mtx)

// копирование верхнего треугольника данных
upperTriangular(mtx) // np.triu(mtx)

// верхнеуровневое копирование
mtx.copy // np.copy(mtx)

// выбрать диагональные элементы матрицы
diag(mtx) // np.diagonal(mtx)
```

Список литературы

1. *Хостаманн К.* Scala для нетерпеливых. – М.: ДМК Пресс, 2013. – 408 с.