cloudera®

Ask Bigger Questions

Developer Training for Spark and Hadoop I: Hands-On Exercises

General Notes

Cloudera's training courses use a Virtual Machine running the CentOS Linux distribution. This VM has CDH (Cloudera's Distribution, including Apache Hadoop) installed in Pseudo-Distributed mode. Pseudo-Distributed mode is a method of running Hadoop whereby all Hadoop daemons run on the same machine. It is, essentially, a cluster consisting of a single machine. It works just like a larger Hadoop cluster, the only key difference (apart from speed, of course!) being that the block replication factor is set to 1, since there is only a single DataNode available.

Getting Started

1. Before starting the exercises, run the course setup script in a terminal window:

```
$ $DEV1/scripts/training_setup_dev1.sh
```

   This script will enable services and set up any data required for the course. You must run this script before starting the Hands-On Exercises.

Working with the Virtual Machine

1. The VM is set to automatically log in as the user `training`. Should you log out at any time, you can log back in as the user `training` with the password `training`.

2. Should you need it, the root password is `training`. You may be prompted for this if, for example, you want to change the keyboard layout. In general, you should not need this password since the `training` user has unlimited sudo privileges.

3. In some command-line steps in the exercises, you will see lines like this:

```
$ hdfs dfs -put shakespeare \
/user/training/shakespeare
```

The dollar sign (**$**) at the beginning of each line indicates the Linux shell prompt. The actual prompt will include additional information (e.g., `[training@localhost workspace]$`) but this is omitted from these instructions for brevity.

The backslash (`\`) at the end of the first line signifies that the command is not completed, and continues on the next line. You can enter the code exactly as shown (on two lines), or you can enter it on a single line. If you do the latter, you should *not* type in the backslash.

Points to note during the exercises

1. The main directory for the exercises for this course is `$DEV1/exercises` (`~/training_materials/dev1/exercises`). Each directory under that one corresponds to an exercise or set of exercises – this is referred to in the instructions as "the exercise directory". Any scripts or files required for the exercise (other than data) are in the exercise directory.

2. Within each exercise directory you may find the following subdirectories:

    a. `solution` – Solution code for each exercise.

    b. `stubs` – A few of the exercises depend on provided starter files containing skeleton code.

    c. Maven project directories – For exercises for which you must write Scala classes, you have been provided with preconfigured Maven project directories. Within these projects are two packages: `stubs`, where you will do your work using starter skeleton classes; and `solution`, containing the solution class.

3. Data files used in the exercises are in `$DEV1DATA` (`~/training_materials/data`). Usually you will upload the files to HDFS before working with them.

4. As the exercises progress, and you gain more familiarity with Hadoop and Spark, we provide fewer step-by-step instructions; as in the real world, we merely give you a requirement and it's up to you to solve the problem! You should feel free to refer to the solutions provided, ask your instructor for assistance, or consult with your fellow students!

5. There are additional bonus exercises for some of the exercises. If you finish the main exercise, please attempt the additional steps.

Catching Up

Most of the exercises in this course build on prior exercises. If you are unable to complete an exercise (even using the provided solution files), or if you need to catch up to the current exercise, run the provided catch up script:

```
$ $DEV1/scripts/catchup.sh
```

The script will prompt for which exercise you are starting; it will set up all the data required as if you had completed all the exercises.

Warning: If you run the catch up script, you will lose all your work! (e.g. All data will be deleted from HDFS, tables deleted from Hive, etc.)

Hands-On Exercise: Collect Web Server Logs with Flume

Files and Data Used in this Exercise

Exercise directory: `$DEV1/exercises/flume`

Data (local): `$DEV1DATA/weblogs/*`

In this exercise, you will configure Flume to ingest web log data from a local directory to HDFS.

Apache web server logs are generally stored in files on the local machines running the server. In this exercise, you will simulate an Apache server by placing provided web log files into a local spool directory, and then use Flume to collect the data.

Both the local and HDFS directories must exist before using the spooling directory source.

Create an HDFS directory for Flume to save ingested data

1. Create a directory in HDFS called `/loudacre/weblogs` to hold the data files Flume ingests, e.g.

```
$ hdfs dfs -mkdir /loudacre/weblogs
```

Create a local directory for web server log output

2. Create the spool directory into which our web log simulator will store data files for Flume to ingest. On the local filesystem create `/flume/weblogs_spooldir`:

```
$ sudo mkdir -p /flume/weblogs_spooldir
```

3. Give all users the permissions to write to the `/flume/weblogs_spooldir` directory:

```
$ sudo chmod a+w -R /flume
```

Configure Flume

In `$DEV1/exercises/flume` under `stubs` create a Flume configuration file with the characteristics listed below. If you need help getting started, you may refer to configuration file in `hints`, or view the solution in the `solutions` directory.

- The source is a spooling directory source that pulls from `/flume/weblogs_spooldir`
- The sink is an HDFS sink that:
  - Writes files to the `/loudacre/weblogs` directory
  - Disables time-based file rolling by setting the `hdfs.rollInterval` property to `0`
  - Disables event-based file rolling by setting the `hdfs.rollCount` property to `0`
  - Sets the `hdfs.rollSize` property as `524288` to enable size-based file rolling at 512KB
  - Writes raw text files (instead of SequenceFile format) by setting `hdfs.fileType` to `DataStream`
- The channel is a Memory Channel that:
  - Can store 10,000 events using the `capacity` property
  - Has a transaction capacity of 10,000 events using the `transactionCapacity` property

Flume Performance Note

Flume's performance on a VM will vary. Sometimes, Flume will exhaust the memory capacity of its channel and give the following error:

```
Space for commit to queue couldn't be acquired. Sinks are
likely not keeping up with sources, or the buffer size is
too tight
```

If you get this error, you will need to increase the `capacity` and `transactionCapacity` properties to a higher value. Then, you will need to restart the agent, clean up any files in HDFS, and repeat the Flume operation.

Run the Agent

Once you have created the configuration file, you need to start the agent and copy the files to the spooling directory.

4. Change directories to the `$DEV1/exercises/flume` directory.

5. Start the Flume agent using the configuration you just made. For example, if you wish to use the solution configuration, enter this command (replace `solution` with `stubs` to run your own configuration):

```
$ flume-ng agent --conf /etc/flume-ng/conf \
--conf-file solution/spooldir.conf \
--name agent1 -Dflume.root.logger=INFO,console
```

6. Wait a few moments for the Flume agent to start up. You will see a message like: `Component type: SOURCE, name: webserver-log-source started`

Simulate Apache web server output

7. Open a separate terminal window, and change to the exercise directory. Run the script to place the web log files in the `/flume/weblogs_spooldir` directory:

```
$ cd $DEV1/exercises/flume
$ ./copy-move-weblogs.sh /flume/weblogs_spooldir
```

   This script will create a temporary copy of the web log files and move them to the `spooldir` directory.

8. Return to the terminal that is running the Flume agent and watch the logging output. The output will give information about the files Flume is putting into HDFS.

9. Once the Flume agent has finished, enter `CTRL+C` to terminate the process.

10. Using the hdfs command line or Hue File Browser, list the files in HDFS that were added by the Flume agent.

    Note that the files that were imported are tagged with a Unix timestamp corresponding to the time the file was imported, e.g. `FlumeData.1427214989392`

Bonus Exercise

Loudacre has a data source that comes in via the network. You want to see some examples since the format is undocumented.

Flume has a `netcat` source, which allows Flume to listen on a port and process the contents received. The address to bind to is specified by the `bind` property, and the port to bind to is specified by the `port` property.

Flume also has a `logger` sink, which logs any incoming data at the `INFO` level in Log4J. This is helpful for debugging and testing Flume configuration. This sink does not require any configuration properties.

Create a Flume configuration file with the following characteristics:

- Uses the `netcat` source to capture data from host `localhost` on port `12345`.
- Uses the `logger` sink
- Uses the `memory` channel

Start the Flume agent using the new configuration file.

In another terminal window, start the test data feed script using the command:

```
$ $DEV1/exercises/flume/script/rng_feed.py
```

Observe the log messages in the Flume agent's terminal and stop the agent after you have seen a few example records.

This is the end of the Exercise

Hands-On Exercise: View the Spark Documentation

Files and Data Used in this Exercise

Exercise directory: None

In this Exercise you will familiarize yourself with the Spark documentation.

1. Start Firefox in your Virtual Machine and visit the Spark documentation on your local machine, using the provided bookmark or opening the URL `file:/usr/lib/spark/docs/_site/index.html`

2. From the **Programming Guides** menu, select the **Spark Programming Guide**. Briefly review the guide. You may wish to bookmark the page for later review.

3. From the **API Docs** menu, select either **Scaladoc** or **Python API**, depending on your language preference. Bookmark the API page for use during class. Later exercises will refer you to this documentation.

This is the end of the Exercise

Hands-On Exercise: Explore RDDs Using the Spark Shell

Files and Data Used in this Exercise

Exercise Directory: `$DEV1/exercises/spark-shell`

Data files (local): `$DEV1DATA/frostroad.txt`

In this Exercise you will start the Spark Shell and read a text file into a Resilient Distributed Data Set (RDD).

You may choose to do this exercise using either Scala or Python. Follow the instructions below for Python, or skip to the next section for Scala.

Note: Instructions for Python are provided in **blue**, while instructions for Scala are in **red**.

Start the Python Spark Shell

1. In a terminal window, start the `pyspark` shell:

```
$ pyspark
```

   You may get several INFO and WARNING messages, which you can disregard. If you don't see the `In[n]>` prompt after a few seconds, hit Return a few times to clear the screen output.

2. Spark creates a SparkContext object for you called `sc`. Make sure the object exists:

```
pyspark> sc
```

> Note: To help you keep track of which shell is being referenced in the instructions, the prompt will be shown here as either **pyspark>** or **scala>**. The actual prompt will vary depending on which version of Python or Scala you are using and what command number you are on.

Pyspark will display information about the `sc` object such as

```
<pyspark.context.SparkContext at 0x2724490>
```

3. Using command completion, you can see all the available SparkContext methods: type `sc.` (`sc` followed by a dot) and then the [TAB] key.

4. You can exit the shell by hitting Ctrl-D or by typing `exit`.

Start and Use the Scala Spark Shell

5. In a terminal window, start the Scala Spark Shell:

```
$ spark-shell
```

You may get several INFO and WARNING messages, which you can disregard. If you don't see the `scala>` prompt after a few seconds, hit Enter a few times to clear the screen output.

6. Spark creates a SparkContext object for you called `sc`. Make sure the object exists:

```
scala> sc
```

> Note: To help you keep track of which shell is being referenced in the instructions, the prompt will be shown here as either **pyspark>** or **scala>**. The actual prompt will vary depending on which version of Python or Scala you are using and what command number you are on.

Scala will display information about the `sc` object such as:

```
res0: org.apache.spark.SparkContext =
org.apache.spark.SparkContext@2f0301fa
```

7. Using command completion, you can see all the available SparkContext methods: type `sc.` (`sc` followed by a dot) and then the [TAB] key.

Load and view text file (Python or Spark)

8. Review the simple text file you will be using by viewing (without editing) the file in a text editor in a separate window (not the Spark shell). The file is located at: `$DEV1DATA/frostroad.txt`.

9. Define an RDD to be created by reading in the test file on the local file system. Use the first command if you are using Python, and the second one if you are using Scala.

```
pyspark> mydata = sc.textFile(\
"file:/home/training/training_materials/\
data/frostroad.txt")
```

```
scala> val mydata = sc.
textFile("file:/home/training/training_materials/data/f
rostroad.txt")
```

(**Note**: In subsequent instructions, both Python and Scala commands will be shown but noted explicitly; Python shell commands are in blue and preceded with `pyspark>`, and Scala shell commands are in red and preceded with `scala>`.)

10. Spark has not yet read the file. It will not do so until you perform an operation on the RDD. Try counting the number of lines in the dataset:

```
pyspark> mydata.count()
```

```
scala> mydata.count()
```

The `count` operation causes the RDD to be materialized (created and populated). The number of lines (23) should be displayed, e.g.

`Out[4]: 23` (Python) *or*
`res0: 23` (Scala)

11. Try executing the `collect` operation to display the data in the RDD. Note that this returns and displays the entire dataset. This is convenient for very small RDDs like this one, but be careful using `collect` for more typical large datasets.

```
pyspark> mydata.collect()
```

```
scala> mydata.collect()
```

12. Using command completion, you can see all the available transformations and operations you can perform on an RDD. Type `mydata.` and then the [TAB] key.

13. You can exit the shell at any time by typing `exit`.

This is the end of the Exercise

Hands-On Exercise: Use RDDs to Transform a Dataset

Files and Data Used in This Exercise:

Exercise Directory: `$DEV1/exercises/spark-transform`

Data files: `/loudacre/weblogs/*` (HDFS)

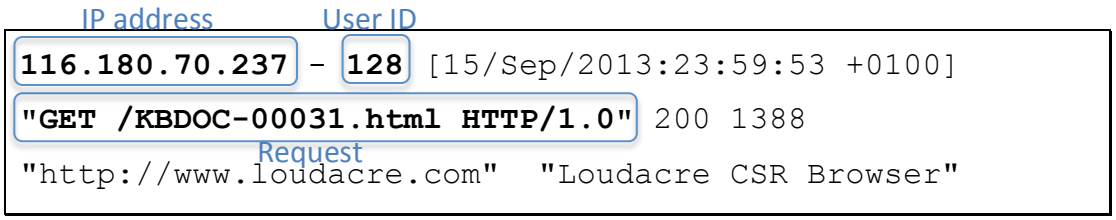
In this Exercise you will practice using RDDs in the Spark Shell.

Use Spark to explore the web server logs you captured in Flume earlier.

Explore the Loudacre web log files

In this section you will be using weblogs you imported into HDFS in the Flume exercise.

1. Using the HDFS command line or Hue File Browser, review one of the files in the HDFS `/loudacre/weblogs` directory, e.g. `FlumeData.1423586038966`. Note the format of the lines, e.g.



2. Set a variable for the data file so you do not have to retype it each time.

```
pyspark> logfile="/loudacre/weblogs/FlumeData.*"
```

```
scala> var logfile="/loudacre/weblogs/FlumeData.*"
```

3. Create an RDD from the data file.

```
pyspark> logs = sc.textFile(logfile)
```

```
scala> var logs = sc.textFile(logfile)
```

4. Create an RDD containing only those lines that are requests for JPG files.

```
pyspark> jpglogs=\
logs.filter(lambda line: ".jpg" in line)
```

```
scala> var jpglogs=
logs.filter(line => line.contains(".jpg"))
```

5. View the first 10 lines of the data using `take`:

```
pyspark> jpglogs.take(10)
```

```
scala> jpglogs.take(10)
```

6. Sometimes you do not need to store intermediate objects in a variable, in which case you can combine the steps into a single line of code. For instance, if all you need is to count the number of JPG requests, you can execute this in a single command:

```
pyspark> sc.textFile(logfile).filter(lambda line: \
  ".jpg" in line).count()
```

```
scala> sc.textFile(logfile).
  filter(line => line.contains(".jpg")).count()
```

7. Now try using the map function to define a new RDD. Start with a simple map that returns the length of each line in the log file.

```
pyspark> logs.map(lambda line: len(line)).take(5)
```

```
scala> logs.map(line => line.length).take(5)
```

This prints out an array of five integers corresponding to the first five lines in the file.

8. That is not very useful. Instead, try mapping to an array of words for each line:

```
pyspark> logs.map(lambda line: line.split()).take(5)
```

```
scala> logs.map(line => line.split(' ')).take(5)
```

This time it prints out five arrays, each containing the words in the corresponding log file line.

9. Now that you know how `map` works, define a new RDD containing just the IP addresses from each line in the log file. (The IP address is the first "word" in each line).

```
pyspark> ips = logs.map(lambda line: line.split()[0])
pyspark> ips.take(5)
```

```
scala> var ips = logs.map(line =>line.split(' ')(0))
scala> ips.take(5)
```

10. Although `take` and `collect` are useful ways to look at data in an RDD, their output is not very readable. Fortunately, though, they return arrays, which you can iterate through:

```
pyspark> for ip in ips.take(10): print ip
```

```
scala> ips.take(10).foreach(println)
```

11. Finally, save the list of IP addresses as a text file:

```
pyspark> ips.saveAsTextFile("/loudacre/iplist")
```

```
scala> ips.saveAsTextFile("/loudacre/iplist")
```

12. In a terminal window or the Hue file browser, list the contents of the `/loudcare/iplist` folder. You should see multiple files, including several `part-xxxxx` files, which are the files containing the output data. ("Part" (partition) files are numbered because there may be results from multiple tasks running on the cluster; you will learn more about this later.) Review the contents of one of the files to confirm that they were created correctly.

Bonus Exercise

Use RDD transformations to create a dataset consisting of the IP address and corresponding user ID for each request for an HTML file. (Disregard requests for other file types). The user ID is the third field in each log file line.

Display the data in the form *`ipaddress/userid`*, e.g.:

```
165.32.101.206/8
100.219.90.44/102
182.4.148.56/173
246.241.6.175/45395
175.223.172.207/4115
…
```

This is the end of the Exercise

Hands-On Exercise: Process Data Files with Spark

Files and Data Used in This Exercise:

Exercise Directory: `$DEV1/exercises/spark-etl`

Data files (local):

`$DEV1DATA/activations/*`

`$DEV1DATA/devicestatus.txt` (Bonus)

Stubs:

`ActivationModels.pyspark`

`ActivationModels.scalaspark`

In this exercise you will parse a set of activation records in XML format to extract the account numbers and model names.

One of the common uses for Spark is doing data Extract/Transform/Load operations. Sometimes data is stored in line-oriented records, like the web logs in the previous exercise, but sometimes the data is in a multi-line format that must be processed as a whole file. In this exercise you will practice working with file-based instead of line-based formats.

Review the API Documentation for RDD Operations

Visit the Spark API page you bookmarked previously. Follow the link at the top for the RDD class and review the list of available methods.

The Data

Review the data in `$DEV1DATA/activations`. Each XML file contains data for all the devices activated by customers during a specific month.

Copy this data to `/loudacre` in HDFS.

Sample input data:

```
<activations>
   <activation timestamp="1225499258" type="phone">
      <account-number>316</account-number>
      <device-id>
          d61b6971-33e1-42f0-bb15-aa2ae3cd8680
      </device-id>
      <phone-number>5108307062</phone-number>
      <model>iFruit 1</model>
   </activation>
   …
</activations>
```

The Task

Your code should go through a set of activation XML files and extract the account number and device model for each activation, and save the list to a file as `account_number:model`.

The output will look something like:

```
1234:iFruit 1
987:Sorrento F00L
4566:iFruit 1
…
```

1. Start with the `ActivationModels` stub script in the exercise directory. (A stub is provided for Scala and Python; use whichever language you prefer.) Note that for convenience you have been provided with functions to parse the XML, as that is not the focus of this Exercise. Copy the stub code into the Spark Shell.

2. Use `wholeTextFiles` to create an RDD from the activations dataset. The resulting RDD will consist of tuples, in which the first value is the name of the file, and the second value is the contents of the file (XML) as a string.

3. Each XML file can contain many activation records; use `flatMap` to map the contents of each file to a collection of XML records by calling the provided `getactivations` function. `getactivations` takes an XML string, parses it, and returns a collection of XML records; `flatMap` maps each record to a separate RDD element.

4. Map each activation record to a string in the format `account-number:model`. Use the provided `getaccount` and `getmodel` functions to find the values from the activation record.

5. Save the formatted strings to a text file in the directory `/loudacre/account-models`.

Bonus Exercise

Another common part of the ETL process is data scrubbing. In this bonus exercise, you will process data in order to get it into a standardized format for later processing.

Review the contents of the file `$DEV1DATA/devicestatus.txt`. This file contains data collected from mobile devices on Loudacre's network, including device ID, current status, location and so on. Because Loudacre previously acquired other mobile provider's networks, the data from different subnetworks has a different format. Note that the records in this file have different field delimiters: some use commas, some use pipes (`|`) and so on. Your task is to

- Load the dataset
- Determine which delimiter to use (hint: the character at position 19 is the first use of the delimiter)
- Filter out any records which do not parse correctly (hint: each record should have exactly 14 values)

- Extract the date (first field), model (second field), device ID (third field), and latitude and longitude (13th and 14th fields respectively)
- The second field contains the device manufacturer and model name (e.g. `Ronin S2`.) Split this field by spaces to separate the manufacturer from the model (e.g. manufacturer `Ronin`, model `S2`.)
- Save the extracted data to comma delimited text files in the `/loudacre/devicestatus_etl` directory on HDFS.
- Confirm that the data in the file(s) was saved correctly.

The solutions to the bonus exercise are in `$DEV1/exercises/spark-etl/bonus`.

This is the end of the Exercise

Hands-On Exercise: Use Pair RDDs to Join Two Datasets

Files and Data Used in This Exercise:

Exercise Directory: `$DEV1/exercises/spark-pairs`

Data files (HDFS):

`/loudacre/weblogs`

`/loudacre/accounts`

In this Exercise you will continue exploring the Loudacre web server log files, as well as the Loudacre user account data, using key-value Pair RDDs.

Explore Web Log Files

Continue working with the web log files, as in the previous exercise.

Tip: In this exercise you will be reducing and joining large datasets, which can take a lot of time. You may wish to perform the exercises below using a smaller dataset, consisting of only a few of the web log files, rather than all of them. Remember that you can specify a wildcard; `textFile("/loudacre/weblogs/*6")` would include only filenames ending with the digit 6.

1. Using map-reduce, count the number of requests from each user.

    a. Use `map` to create a Pair RDD with the user ID as the key, and the integer 1 as the value. (The user ID is the third field in each line.) Your data will look something like this:

(*userid*,1)
(*userid*,1)
(*userid*,1)
…

b. Use `reduce` to sum the values for each user ID. Your RDD data will be similar to:

(*userid*,5)
(*userid*,7)
(*userid*,2)
…

2. Use `countByKey` to determine how many users visited the site for each frequency. That is, how many users visited once, twice, three times and so on.

   a. Use `map` to reverse the key and value, like this:

(5,*userid*)
(7,*userid*)
(2,*userid*)
…

   b. Use the `countByKey` action to return a Map of *frequency*:*user-count* pairs.

3. Create an RDD where the user id is the key, and the value is the list of all the IP addresses that user has connected from. (IP address is the first field in each request line.)

   - Hint: Map to `(userid,ipaddress)` and then use `groupByKey`.

(*userid*,20.1.34.55)
(*userid*,245.33.1.1)
(*userid*,*65.50.196.141*)
…

(*userid*,[20.1.34.55, 74.125.239.98])
(*userid*,[75.175.32.10, 245.33.1.1, 66.79.233.99])
(*userid*,[65.50.196.141])
…

Join Web Log Data with Account Data

In the Sqoop exercise you completed earlier, you imported data files containing Loudacre's customer account data from MySQL to HDFS. Review that data now (located in `/loudacre/accounts`). The first field in each line is the user ID, which corresponds to the user ID in the web server logs. The other fields include account details such as creation date, first and last name and so on.

4. Join the accounts data with the weblog data to produce a dataset keyed by user ID which contains the user account information and the number of website hits for that user.

    a. Create an RDD based on the accounts data consisting of key/value-array pairs: `(userid,[values…])`

(*userid1*,[*userid1*,2008-11-24 10:04:08,\N,Cheryl,West,4905 Olive Street,San Francisco,CA,…])
(*userid2*,[*userid2*,2008-11-23 14:05:07,\N,Elizabeth,Kerns,4703 Eva Pearl Street,Richmond,CA,…])
(*userid3*,[*userid3*,2008-11-02 17:12:12,2013-07-18 16:42:36,Melissa,Roman,3539 James Martin Circle,Oakland,CA,…])
…

    b. Join the Pair RDD with the set of user-id/hit-count pairs calculated in the first step.

(*userid1*,([*userid1*,2008-11-24 10:04:08,\N,Cheryl,West,4905 Olive Street,San Francisco,CA,…],**4**))
(*userid2*,([*userid2*,2008-11-23 14:05:07,\N,Elizabeth,Kerns,4703 Eva Pearl Street,Richmond,CA,…],**8**))
(*userid3*,([*userid3*,2008-11-02 17:12:12,2013-07-18 16:42:36,Melissa,Roman,3539 James Martin Circle,Oakland,CA,…],**1**))
…

c. Display the user ID, hit count, and first name (3rd value) and last name (4th value) for the first 5 elements, e.g.:

```
userid1 4 Cheryl West
userid2 8 Elizabeth Kerns
userid3 1 Melissa Roman
…
```

Bonus Exercises

If you have more time, attempt the following challenges:

1. Challenge 1: Use `keyBy` to create an RDD of account data with the postal code (9th field in the CSV file) as the key.
    - Tip: Assign this new RDD to a variable for use in the next challenge
2. Challenge 2: Create a pair RDD with postal code as the key and a list of names (Last Name,First Name) in that postal code as the value.
    - Hint: First name and last name are the 4th and 5th fields respectively
    - Optional: Try using the `mapValues` operation
3. Challenge 3: Sort the data by postal code, then for the first five postal codes, display the code and list the names in that postal zone, e.g.

```
--- 85003
Jenkins,Thad
Rick,Edward
Lindsay,Ivy
…
--- 85004
Morris,Eric
Reiser,Hazel
Gregg,Alicia
Preston,Elizabeth
…
```

This is the end of the Exercise

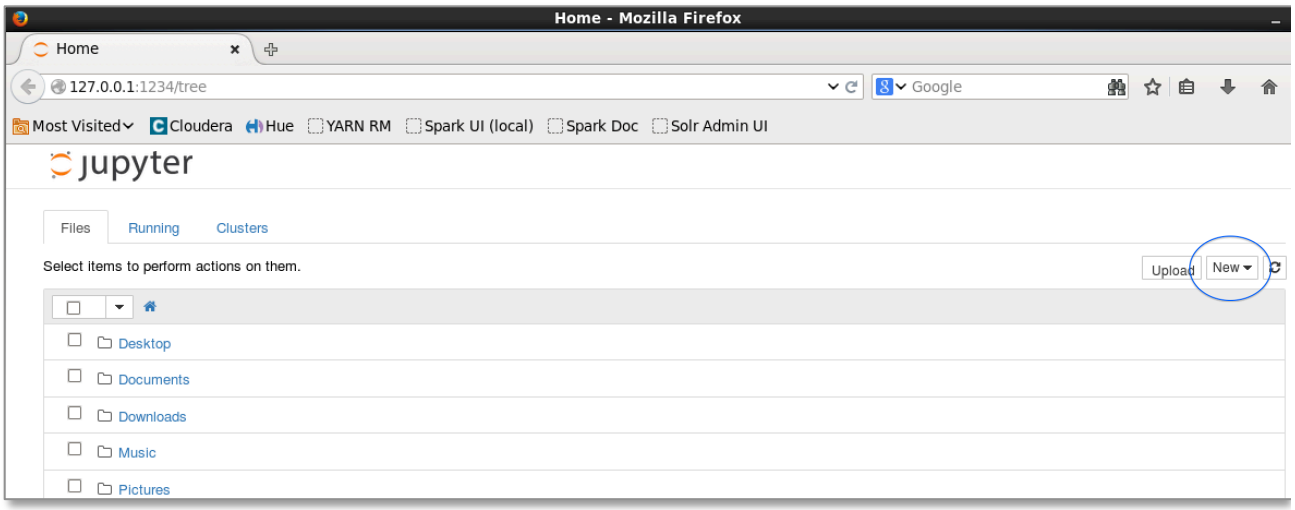
Appendix A: Enabling iPython Notebook

iPython Notebook is installed on the VM for this course. To use it instead of the command-line version of iPython, follow these steps:

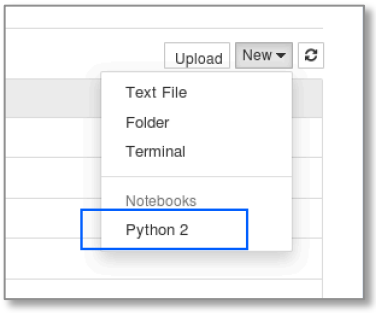
1. Open the following file for editing: `/home/training/.bashrc`
2. Uncomment out the following line (remove the leading #).

```
# export PYSPARK_DRIVER_PYTHON_OPTS='notebook ……..jax'
```

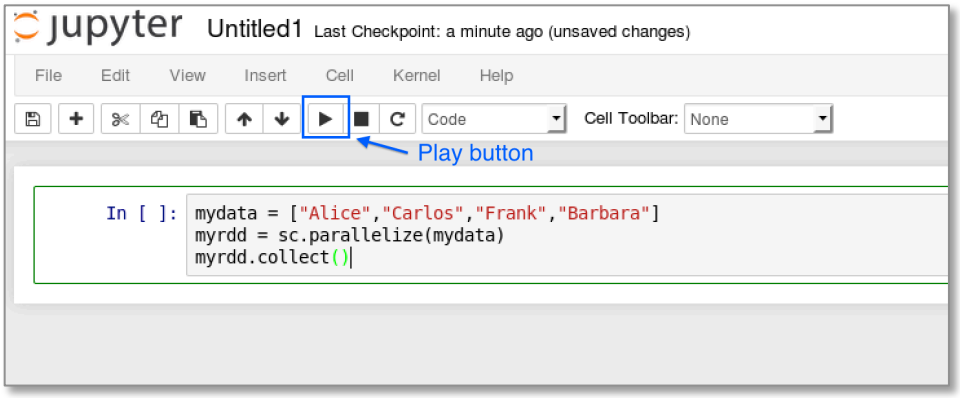
3. Save the file.
4. Open a new terminal window. (Must be a new terminal so it loads your edited `.bashrc` file).
5. Enter `pyspark` in the terminal. This will cause a browser window to open, and you should see the following web page:



6. On the right hand side of the page select **Python 2** from the **New** menu



7. Enter some spark code such as the following and use the play button to execute your spark code.



8. Notice the output displayed.