ITECH2302 Big Data Management Laboratory - Spark

Objectives:

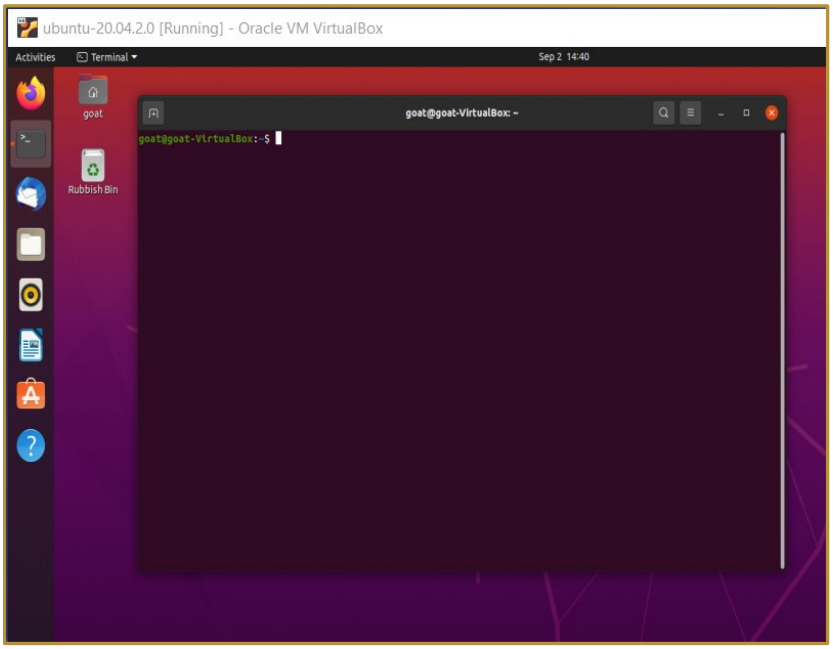
- Introduction to Spark
- Jupyter notebooks overview (you have already achieved this)
- pySpark

Activity 1

Apache Hadoop

1. Start Apache Hadoop

Open a terminal with the ubuntu operating system 



Write the following commands:

```
$ ssh localhost
$ hdfs namenode -format
$ start-dfs.sh
$ start-yarn.sh
```

If the output from:

```
$ jps
```

..doesn't look like the following,

```
5042 DataNode
5299 SecondaryNameNode
4888 NameNode
```

5516 ResourceManager

5677 NodeManager

6046 Jps

Then maybe the *datanode* didn't start correctly because it was left in a corrupted state. This is easy to fix by using the following commands:

```
$ stop-all.sh
$ rm -rf /home/goat/hadoopdata/hdfs/datanode/*
$ start-all.sh
```

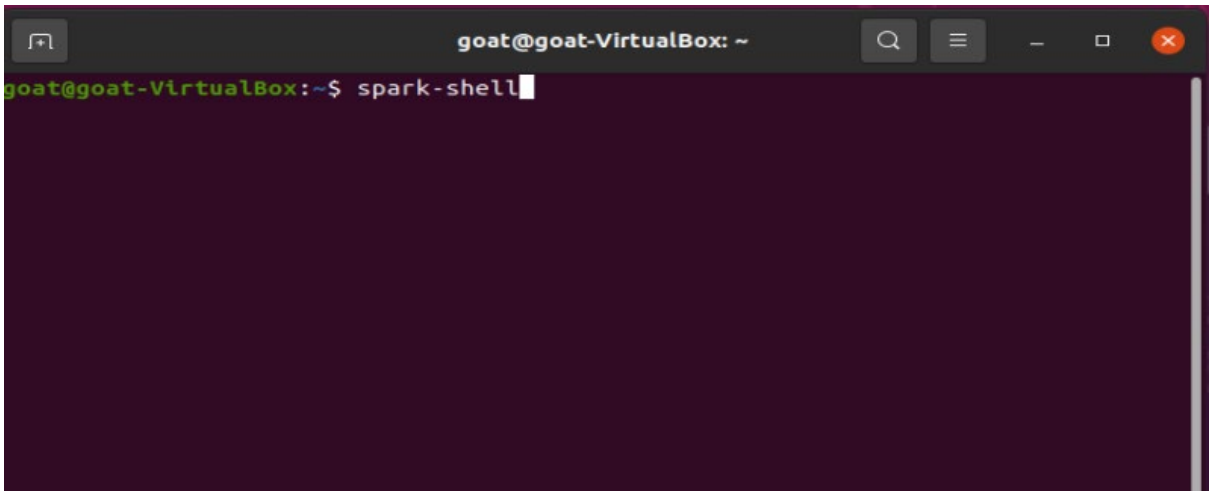
Try `jps` again, you should see the datanode listed now.

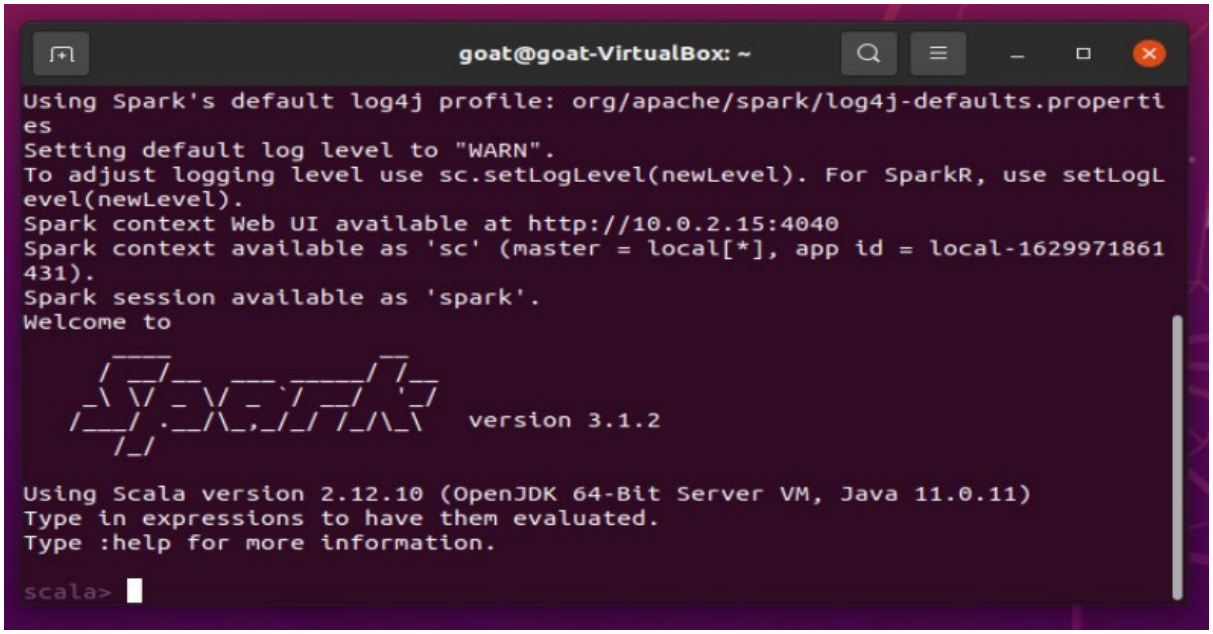
Activity 2

Scala

Write the following commands to open up the scala shell:

$ spark-shell





And to shut down the scala shell:

scala> sys.exit

```
scala> sys.exit
```

There are some datafiles that can be used here:
/home/goat/hadoop_spark/spark/lab_data/spark_data/accounts.csv
/home/goat/hadoop_spark/spark/lab_data/spark_data/rdds/blog.txt

Try and type in some scala programs from the following resources:

- https://spark.apache.org/examples.html
- https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples

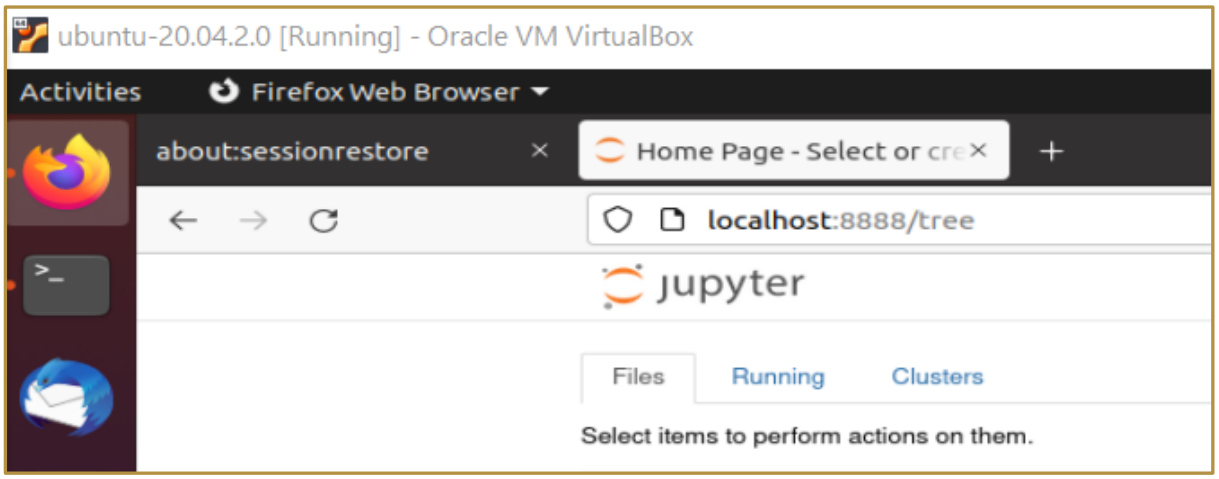
Activity 3

Getting started with Python and Jupyter notebooks

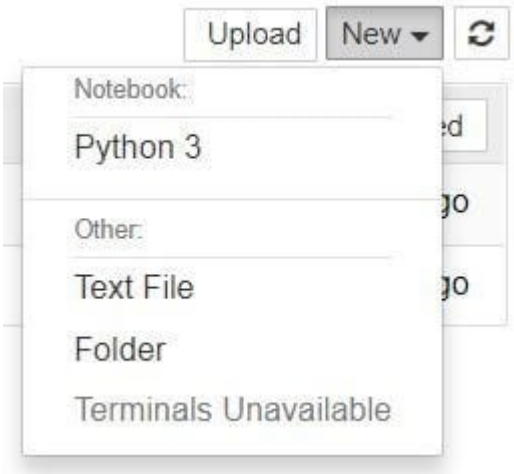
Write the following commands to open up python notebooks:
$ pyspark

```
goat@goat-VirtualBox:~$ pyspark
```

Firefox should open at the following URL. This is where you can upload your python notebooks (with extension .pynb)
It is also possible to start the dashboard on any system via the command prompt (or terminal on Unix systems) by entering the command jupyter notebook; in this case, the current working directory will be the start-up directory.
With Jupyter Notebook open in your browser, you may have noticed that the URL for the dashboard is something like https://localhost:8888/tree. Localhost is not a website, but indicates that the content is being served from your *local* machine: your own computer.
Jupyter's Notebooks and dashboard are web apps, and Jupyter starts up a local Python server to serve these apps to your web browser, making it essentially platform-independent and opening the door to easier sharing on the web.



The dashboard's interface is mostly self-explanatory — browse to the folder in which you would like to create your first notebook, click the "New" drop-down button in the top-right and select "Python 3":



Your Jupyter Notebook will open in new tab — each notebook uses its own tab because you can open multiple notebooks simultaneously. If you switch back to the dashboard, you will see the new file Untitled.ipynb and you should see some green text that tells you your notebook is running.

What is an ipynb File?

The short answer: each .ipynb file is one notebook, so each time you create a new notebook, a new .ipynb file will be created.

The longer answer: Each .ipynb file is a text file that describes the contents of your notebook in a format called JSON. Each cell and its contents, including image attachments that have been converted into strings of text, is listed therein along with some metadata.

You can edit this yourself — if you know what you are doing! — by selecting "Edit > Edit Notebook Metadata" from the menu bar in the notebook. You can also view the contents of your notebook files by selecting "Edit" from the controls on the dashboard. However, in most cases, there's no reason you should ever need to edit your notebook metadata manually.

The Notebook Interface

Now that you have an open notebook in front of you, its interface will hopefully not look entirely alien. After all, Jupyter is essentially just an advanced word processor.

Check out the menus to get a feel for it, especially take a few moments to scroll down the list of commands in the command palette, which is the small button with the keyboard icon (or Ctrl + Shift + P).

There are two fairly prominent terms that you should notice, which are probably new to you: *cells* and *kernels* are key both to understanding Jupyter and to what makes it more than just a word processor. Fortunately, these concepts are not difficult to understand.

- A **kernel** is a "computational engine" that executes the code contained in a notebook document.
- A **cell** is a container for text to be displayed in the notebook or code to be executed by the notebook's kernel.

Cells

Cells form the body of a notebook. In the screenshot of a new notebook in the section above, that box with the green outline is an empty cell. There are two main cell types that we will cover:

- A **code cell** contains code to be executed in the kernel. When the code is run, the notebook displays the output below the code cell that generated it.
- A **Markdown cell** contains text formatted using Markdown and displays its output in-place when the Markdown cell is run.
-

The first cell in a new notebook is always a code cell.
Let's test it out with a classic hello world example: Type print('Hello World!') into the cell and click the run button 

in the toolbar above or press Ctrl + Enter.
The result should look like this:

```
print('Hello World!')

Hello World!
```

When we run the cell, its output is displayed below and the label to its left will have changed from In [ ] to In [1].
The output of a code cell also forms part of the document. You can always tell the difference between code and Markdown cells because code cells have that label on the left and Markdown cells do not.

The "In" part of the label is simply short for "Input," while the label number indicates *when* the cell was executed on the kernel — in this case the cell was executed first.

Run the cell again and the label will change to In [2] because now the cell was the second to be run on the kernel.

From the menu bar, click *Insert* and select *Insert Cell Below* to create a new code cell underneath your first and try out the following code to see what happens. Do you notice anything different?

```
import time


time.sleep(3)
```

This cell doesn't produce any output, but it does take three seconds to execute. Notice how Jupyter signifies when the cell is currently running by changing its label to `In [*]`.

In general, the output of a cell comes from any text data specifically printed during the cell's execution, as well as the value of the last line in the cell, be it a lone variable, a function call, or something else. For example:

```
def say_hello(recipient):

    return 'Hello, {}!'.format(recipient)




say_hello('Tim')

'Hello, Tim!'
```

You'll find yourself using this almost constantly in your own projects.

Kernels

Behind every notebook runs a kernel. When you run a code cell, that code is executed within the kernel. Any output is returned back to the cell to be displayed. The kernel's state persists over time and between cells — it pertains to the document as a whole and not individual cells.

For example, if you import libraries or declare variables in one cell, they will be available in another. Let's try this out to get a feel for it. First, we'll import a Python package and define a function:

```
import numpy as np

def square(x):


    return x * x
```

Once we've executed the cell above, we can reference `np` and `square` in any other cell.

```
x = np.random.randint(1, 10)

y = square(x)


print('%d squared is %d' % (x, y))
```

```
1 squared is 1
```

This will work regardless of the order of the cells in your notebook. As long as a cell has been run, any variables you declared or libraries you imported will be available in other cells.

You can try it yourself, let's print out our variables again.

```
print('Is %d squared %d?' % (x, y))
```

```
Is 1 squared 1?
```

No surprises here! But what happens if we change the value of y?

```
y = 10

print('Is %d squared is %d?' % (x, y))
```

If we run the cell above, what do you think would happen?

We will get an output like: Is 4 squared 10?. This is because once we've run the y = 10 code cell, y is no longer equal to the square of x in the kernel.

Most of the time when you create a notebook, the flow will be top-to-bottom. But it's common to go back to make changes. When we do need to make changes to an earlier cell, the order of execution we can see on the left of each cell, such as In [6], can help us diagnose problems by seeing what order the cells have run in.

And if we ever wish to reset things, there are several incredibly useful options from the Kernel menu:

- Restart: restarts the kernel, thus clearing all the variables etc that were defined.
- Restart & Clear Output: same as above but will also wipe the output displayed below your code cells.
- Restart & Run All: same as above but will also run all your cells in order from first to last.

If your kernel is ever stuck on a computation and you wish to stop it, you can choose the Interrupt option.

Naming Your Notebooks

Before you start writing your project, you'll probably want to give it a meaningful name. file name Untitled in the upper left of the screen to enter a new file name, and hit the Save icon (which looks like a floppy disk) below it to save. Note that closing the notebook tab in your browser will **not** "close" your notebook in the way closing a document in a traditional application will. The notebook's kernel will continue to run in the background and needs to be shut down before it is truly "closed" — though this is pretty handy if you accidentally close your tab or browser!

If the kernel is shut down, you can close the tab without worrying about whether it is still running or not.
The easiest way to do this is to select "File > Close and Halt" from the notebook menu. However, you can also shutdown the kernel either by going to "Kernel > Shutdown" from within the notebook app or by selecting the notebook in the dashboard and clicking "Shutdown" (see image below).

Setup

It's common to start off with a code cell specifically for imports and setup, so that if you choose to add or change anything, you can simply edit and re-run the cell without causing any side-effects.

```
%matplotlib inline

import pandas as pd

import matplotlib.pyplot as plt


import seaborn as sns sns.set(style="darkgrid")
```

We'll import pandas to work with our data, Matplotlib to plot charts, and Seaborn to make our charts prettier. It's also common to import NumPy but in this case, pandas imports it for us.
That first line isn't a Python command, but uses something called a line magic to instruct Jupyter to capture Matplotlib plots and render them in the cell output.

For now, let's go ahead and load our data.

```
df = pd.read_csv('fortune500.csv')
```

It's sensible to also do this in a single cell, in case we need to reload it at any point.

Save and Checkpoint
It is best practice to save regularly. Pressing Ctrl + S will save our notebook by calling the "Save and Checkpoint" command - every time we create a new notebook, a checkpoint file is created along with the notebook file. It is located within a hidden subdirectory of your save location called .ipynb_checkpoints and is also a .ipynb file.

By default, Jupyter will autosave your notebook every 120 seconds to this checkpoint file without altering your primary notebook file. When you "Save and Checkpoint," both the notebook and checkpoint files are updated. Hence, the checkpoint enables you to recover your unsaved work in the event of an unexpected issue.
You can revert to the checkpoint from the menu via "File > Revert to Checkpoint."

Here are some datafiles you can use, or download something from the Internet:

/home/goat/hadoop_spark/spark/lab_data/spark_data/accounts.csv

/home/goat/hadoop_spark/spark/lab_data/spark_data/rdds/blog.txt

Create a new python notebook. Find a textfile to read, e.g.:

```
textfile = sc.textFile("blog.txt")
```

This will create an RDD containing one entry per line in the file.

To take a look at first 10 entries in RDD type:

```
textfile.take(10)
```

Now, count all the words in this text file.

```
counts = textfile
.flatMap(lambda line:line.split(" "))
.map(lambda word: (word,1))
.reduceByKey(lambda a, b: a + b)
```

To take a look on the result:

```
counts.take(5)
```

This command will show you first five entries in result.

Activity 4

Run and examine the following ipynb files in the spark directory:

The following notebooks come from https://github.com/jadianes/spark-py-notebooks and can be found in this folder: /home/goat/hadoop_spark/hadoop/lab_data/ spark-py-notebooks-master

Work your way through the notebooks:

- nb1-rdd-creation
- nb2-rdd-basics
- nb3-rdd-sampling
- nb4-rdd-set
- nb5-rdd-aggregations
- nb6-rdd-key-value
- nb7-mllib-statistics
- nb8-mllib-logit
- nb9-mllib-trees
- nb10-sql-dataframes

Activity 5

Google BigQuery

You might want to check out Google BigQuery:

- https://cloud.google.com/bigquery

And its provisions for JSON in its query language:

- https://cloud.google.com/bigquery/docs/reference/standard-sql/json_functions
- https://docs.snowflake.com/en/sql-reference/functions/parse_json.html