Big Data Infrastructures & Technology - Assignment 1

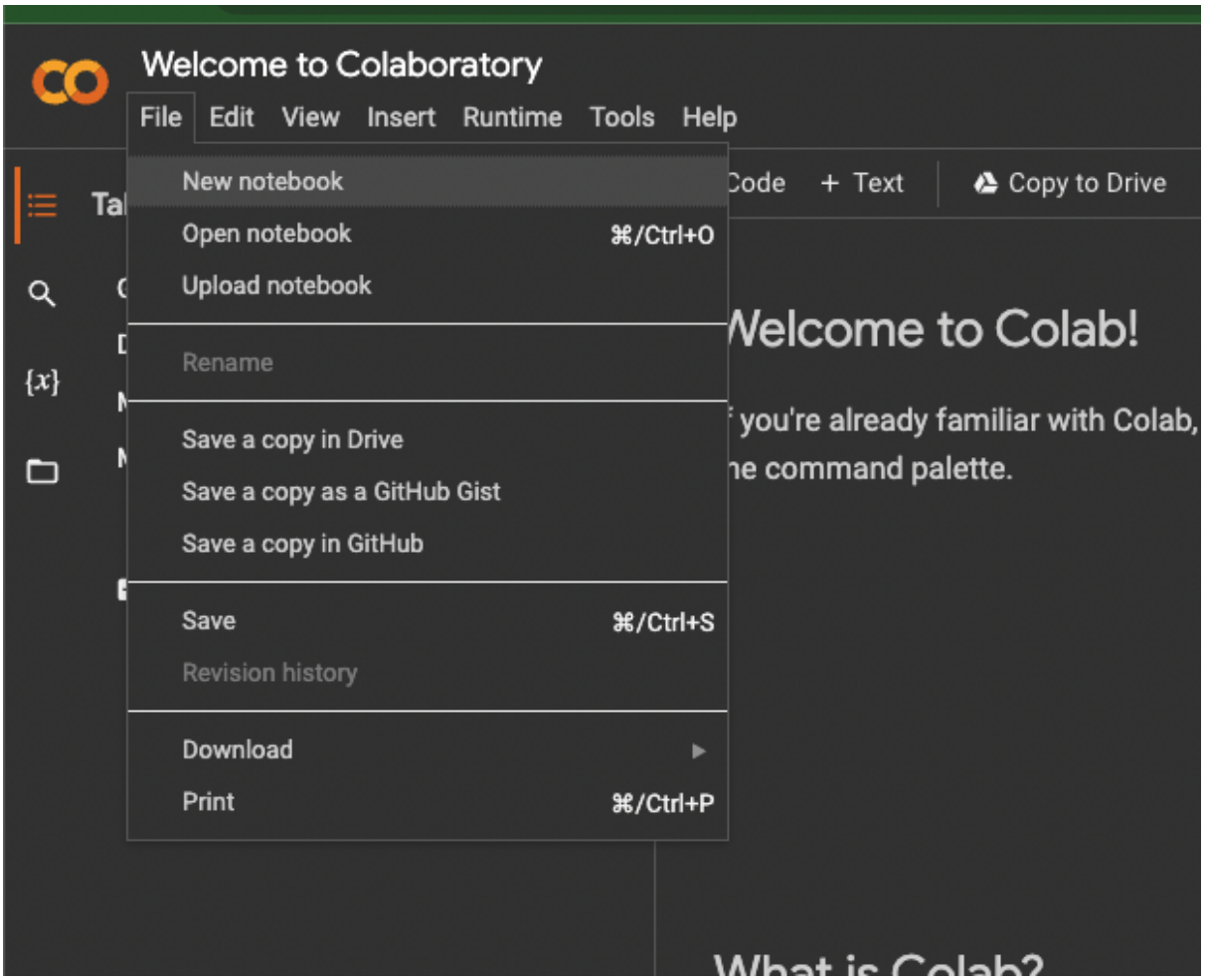
The first practicum is an introductory assignment. The goal of this assignment is to learn how to use Python and Spark, and to familiarize yourself with the Jupyter Notebook environment. These tools will be used in all the other practical assignments.

Google Colab

While you can run Jupyter notebooks (or Python) on your own machine - the easiest way of setting this up is in your browser using Google Colab: https://colab.research.google.com/

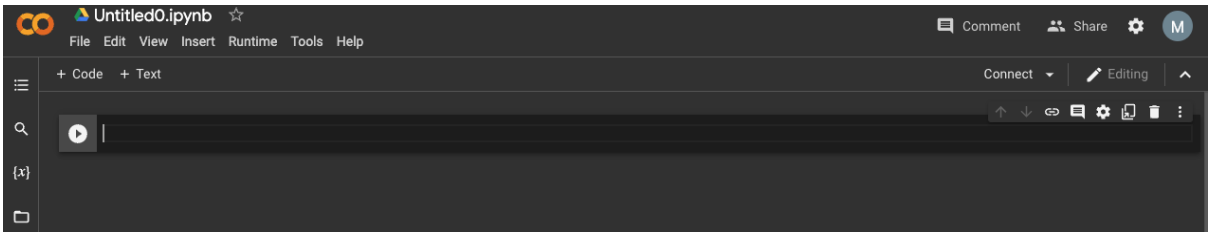
Navigate to Google Colab and log into your google account.

After that, you will be able to create a new Jupyter notebook.



Basic Usage

After creating a new notebook, you will see the following screen. This is the main notebook screen. Notebooks allow you to run Python code directly in your browser.



Try typing the following code and executing it by pressing `Shift+Enter`.

```
for i in range(1, 10):
    print(i)
```

This will print a list of all numbers from 1 to 9.

Basic Data Wrangling & Analysis

We will start doing some basic data wrangling and analysis. For this task, we will work with the complete works of William Shakespeare. First, we need to load this dataset into our Notebook.

Download the works of Shakespeare onto your local machine.
These can be found at the following url:
https://raw.githubusercontent.com/bbejeck/hadoop-algorithms/master/src/shakespeare.txt

You can download the file using the "wget" command. You can execute arbitrary shell commands by placing an exclamation mark before the command:

```
!wget
https://raw.githubusercontent.com/bbejeck/hadoop-algorithms/master
/src/shakespeare.txt
```

We can verify that the file is there by running the "head" command to read the first few lines of the file:

```
!head shakespeare.txt
```

This should produce the first few lines of the file.

Now we are ready to start our analysis. We will analyze the data in two different ways. First, we will perform a **local analysis** where we will use a simple Python script to analyze this data set. Afterwards, we will perform a **cluster analysis** where will use **Spark** to analyze the data.

You can read more about **pyspark** here**:**
https://spark.apache.org/docs/latest/rdd-programming-guide.html

The analysis we are going to perform is very simple: we want to find the top 100 most common words in the works of Shakespeare.

Local Analysis

In the local analysis, we will only use regular Python to operate on the data. First, we will load the data from the file into our Python session.

```
with open('shakespeare.txt', 'r') as f:
    text = f.read()
```

This will load the complete contents of the file into a local variable called `text`. We can verify that the loading was successful by printing the contents of the `text` variable.

```
print(text[:100])
```

This should print the following text:

```
The Project Gutenberg EBook of The Complete Works of William
Shakespeare, by William Shakespeare
```

Now we are going to count the frequency of each of the words in the document. First, we split up the document into words by splitting the text on the space character. We also filter out empty words using the `len(x) > 0` filter.

```
words = [x for x in text.replace('\n', ' ').split(' ') if len(x) >
0]
```

Then we perform the counting of words. We create a dictionary that maps from `Word → Count` (i.e. for every word, it contains the amount of times that word occurs in the text). We iterate over the words in the text and, for each word in the text, increase its count by one.

```
word_count = {}
for word in words:
    if word not in word_count:
        word_count[word] = 0
    word_count[word] = word_count[word] + 1
```

This gives us a dictionary with the word count for each of the words. Now all we need to do is extract the top 100 most frequently occurring words (i.e. the top 100 words with the highest count in the dictionary). We do this by converting the dictionary into a list using the `list` function, and then sorting it by the inverse count.

```
word_count_list = list(word_count.items())
sorted_words = sorted(word_count_list, key=lambda x: -x[1])
```

We can then print the top 100 words by iterating over the sorted list:

```
for i in range(100):
    print(sorted_words[i])
```

This results in the following result:

```
('the', 23407)
('I', 19540)
…
```

Cluster Analysis

For the cluster analysis, we are going to use **pyspark** to execute the code. In this case, we will run **pyspark** in local mode (i.e. running it only on a single machine). However, the methods that we describe here can be scaled to run on multiple machines. First, we need to install pyspark using the pip command:

```
!pip install pyspark
```

Then we can import the spark module, and initialize the Spark context.

```
from pyspark import SparkConf, SparkContext
conf =
SparkConf().setMaster("local").setAppName("test").set('spark.ui.enabl
ed','false')

sc = SparkContext.getOrCreate(conf = conf)
```

We load the shakespeare text file into Spark using the `textFile` command.

```
shakespeare = sc.textFile("shakespeare.txt")
```

This command gives us a Spark RDD (Resilient Distributed Dataset). In our case, the RDD resides on our local machine, as we are running Spark in local mode. However, the same object type can refer to a dataset that is distributed over a cluster of machines as well. We can learn more about the operations that we can perform on a RDD with the `help` command:

```
help(shakespeare)
```

In our case, we want to split up the text file into separate words, count the frequency of each of the words and extract the most frequently occurring words. We start out by splitting the RDD into separate words:

```
words = shakespeare.flatMap(lambda line: line.split(' '))
```

Note that this command does not actually execute anything yet. Spark commands only execute when the actual result of the operation is requested by the client. We can force Spark to execute this command by retrieving the result of the operation:

```
words.collect()
```

This should print the set of split up words. Now we will perform the actual counting. First, we want to filter out the empty words. We do this using the `filter` operation.

```
filtered_words = words.filter(lambda x: len(x) > 0)
```

Then we transform every word into a tuple `(word, 1)` using the `map` function. The value `1` represents how many times the word occurs in the text. We then use the `reduceByKey` function to merge the tuples together by their key (the `word`). This results in the set of `(word, count)` tuples.

```
tuples = filtered_words.map(lambda word: (word, 1))
counts = tuples.reduceByKey(lambda a, b: a + b)
```

Again, these operations do not result in any execution taking place yet. We can inspect any of the intermediate structures using the `collect()` function to verify that they are what we expect them to be.

Finally, we extract the top 100 counts using the `takeOrdered` function. This will result in the actual computation taking place, and retrieving the final set of the most frequently occurring words.

```
results = counts.takeOrdered(100, key=lambda x: -x[1])
for entry in results:
     print(entry)
```

This results in the following counts:

```
('the', 23407)
('I', 19540)
('and', 18358)
('to', 15682)
...
```

Plotting Data Using Plotly

After we have performed our analysis and figured out the top words, we typically want to create a visualization of this data. In this example, we will use the **plotly** library to create a pie diagram of the most frequently occurring words.

More information about **plotly** can be found here: https://plot.ly/python/

First we have to install plotly. We can do this using pip again.

```
!pip install plotly
```

Then we need to initialize the `plotly` library. We can do that using the following import statements:

```
from plotly.offline import init_notebook_mode,iplot
import plotly.graph_objs as go
init_notebook_mode(connected=True)
```

This sets us up for plotting our results inside our notebook. Now we need to prepare our data for plotting. We have to divide our results into a set of `labels` and a set of `values`. In our case, the `labels` for the pie chart are the words and the values are the `counts` of the respective words. We can extract these from the result we obtained in **Cluster Analysis** is follows:

```
labels = [x[0] for x in results[:10]]
values = [x[1] for x in results[:10]]
```

We can then create a plot using the `go.Figure` and `go.Pie` commands:

```
go.Figure(go.Pie(labels=labels,
values=values)).show(renderer="colab")
```

Bonus Task

The words that we have found as the most common words are not very surprising or interesting. In any text, articles such as *"the"* occur very frequently. Another problem with our analysis is that we did not perform any proper normalization of the words. Both the words *"The"* and *"the"* (with different capitalization) occur in our result, whereas it would make more sense if they would count as the same word.

For the bonus assignment the following two tasks should be done before to performing the count:

1. Perform simple normalization of words (converting everything to the same case and filter out punctuation).
   (*Note that in real analysis more complex normalization is typically performed, such as reducing plural words to their singular form)*
2. Remove stop words (such as *the, and, to, …*) from the set of words that is counted.
   (A list of common stop words can be found here: https://gist.githubusercontent.com/Mytherin/413848e4a55598263ed58eaf4f15210f/raw/e4c403ade8b9fb0db1a9e37ee03fc9f77934b881/List%2520of%2520Stop%2520Words%2520in%2520Python)

After performing these operations on the data, plot the top 10 most frequently occurring words again.