# PySpark HW report

## Main task

Task: Create a map reduce word count application and run it

Project Structure:

```
main-task
├── main.py # Python script
├── target # Result directory, ignored in .gitignore
└── wnp.txt # War and Peace book text
```

Firstly, I installed *PySpark* via running `pip3 install pyspark`

Then I created a project and started exploring *PySpark* without any tutorials 😄

I didn't save the exact code I wrote but it was somewhat similar to final version:

```
def other_solution(self):
    '''
        My custom solution
    '''

    self.spark.read.text(INPUT_PATH).withColumn(
        'word',
        f.explode(
            f.split(
                # Splitting strings and getting separate words
                f.lower(f.col('value')),
                ' ',
            )
        )
    ).filter(
        # Filtering empty strings
        f.col('value') != '',
    ).groupBy(
        'word',
    ).count().sort(
        # Counting and sorting by count descending
        'count',
        ascending=False,
    ).show(n=WORDS_NUMBER)
```

Next I ran command `python3 main.py` and saw the following output:

```
+----+-----+
|word|count|
+----+-----+
| the|34270|
| and|21392|
|  to|16504|
|  of|14909|
|   a|10387|
|  he| 9298|
|  in| 8737|
| his| 7930|
|that| 7410|
| was| 7205|
+----+-----+
only showing top 10 rows
```

Quite good, however, I didn't use Reduce, so I checked out the tutorial and wrote next code based on it:

```
def main_solution(self):
    '''
        Solution based on presentation tutorial
    '''

    res = self.spark.sparkContext.textFile(
        self.input_path,
    ).flatMap(
        # Splitting words
        lambda line: line.split(' '),
    ).filter(
        # Filtering words: for instance, 'abc' is a word and 'abc123.,-'
is not
        lambda line: IS_WORD.match(line),
    ).map(
        lambda word: (word, 1),
    ).reduceByKey(
        lambda count1, count2: count1 + count2,
    ).sortBy(
        # Sorting by count descending
        lambda pair: pair[1],
        ascending=False,
    )

    res.saveAsTextFile(self.output_path)
    print(
        *map(lambda pair: f'{pair[0]}: {pair[1]}', res.collect()
[:WORDS_NUMBER]),
        sep='\n',
    )
```

So I ran the app and got target directory with all the words counts:

```
...
('from', 2517)
('you', 2422)
('said', 2406)
('were', 2352)
('by', 2316)
...
('scented,', 1)
('canceled.', 1)
('wearisome.”', 1)
('wound-up', 1)
('Novosíltsev’s', 1)
...
```

Also I saw top 10 words were written to terminal:

```
the: 31714
and: 20560
to: 16324
of: 14860
a: 10017
in: 8232
he: 7631
his: 7630
that: 7228
was: 7193
```

As you can see, the results are a bit different because in my solution I only filtered empty strings, while in the main one I wrote regular expression which checks that all of the symbols are English alphabet letters

Now we used MapReduce and are ready to work with *Spark*: `spark-submit main.py`

*Spark* when it just started:


main-1

*Spark* when it finished working:


main-2

There are so many jobs because I am running both of the solutions

## Extra task

Task: Launch *Spark* cluster containing 1 master and 2 workers and run the previous app on it

Project structure

```
extra-task
├── docker-compose.yml # File for creating 3 containers
├── input # Directory with needed file
│   └── wnp.txt
├── output # Directory with all the output
│   ├── worker-a
│   └── worker-b
└── src # Directory with script
    └── main.py
```

To be honest, launching virtual machines is a quite tedious process, so I decided to automise it with *Docker*

The whole script of *Compose*:

```
version: "3.3"
services:

  spark-master:
    image: apache/spark-py
    user: root
    ports:
      - "4040:8080"
    volumes:
      - ./input:/input:ro
      - ./src:/src
    entrypoint: /opt/spark/sbin/start-master.sh
    environment:
      - SPARK_NO_DAEMONIZE=1

  spark-worker-a:
    image: apache/spark-py
    user: root
    ports:
      - "4041:8081"
    volumes:
      - ./input:/input:ro
      - ./output/worker-a:/output
    entrypoint: /opt/spark/sbin/start-worker.sh spark://spark-master:7077
    environment:
      - SPARK_NO_DAEMONIZE=1

  spark-worker-b:
    image: apache/spark-py
    user: root
    ports:
      - "4042:8081"
    volumes:
      - ./input:/input:ro
      - ./output/worker-b:/output
    entrypoint: /opt/spark/sbin/start-worker.sh spark://spark-master:7077
```

```
    environment:
      - SPARK_NO_DAEMONIZE=1
```

To launch all 3 containers we need to run `docker-compose up --build -d`

After it finished working you can see output:

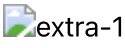
```
[+] Running 4/4
 ⠿ Network extra-task_default             Created   0.0s
 ⠿ Container extra-task-spark-master-1    Started   0.7s
 ⠿ Container extra-task-spark-worker-a-1  Started   0.9s
 ⠿ Container extra-task-spark-worker-b-1  Started   0.9s
```

To stop the whole application we need to execute command `docker-compose down`

It will be followed by logs:

```
[+] Running 4/3
 ⠿ Container extra-task-spark-worker-b-1  Removed   10.3s
 ⠿ Container extra-task-spark-worker-a-1  Removed   10.4s
 ⠿ Container extra-task-spark-master-1    Removed   10.4s
 ⠿ Network extra-task_default             Removed   0.1s
```

Furthermore, we can see stats via *Docker Desktop* graphical user interface:


extra-1

Now it is time to check if the *Spark* application is running

We can visit *localhost:4040* to check the state of master, *localhost:4041* and *localhost:4042* to see the workers:


extra-2


extra-3


extra-4

So we have the whole system running, now it is time to submit the script

To do it we need to connect to container terminal via `docker exec -it <container-id> bash` or simply use the *Docker Desktop* application

As soon as we reach the terminal, we need to run the following command to make the whole system to start working `cd / && ./opt/spark/bin/spark-submit src/main.py`:


extra-5

Afterwards we can look through the logs (they were actually the same as the previous task except more *Spark* info logs) and finally visit *localhost:4040*, *localhost:4041* and *localhost:4042* once more:


extra-6


extra-7


extra-8

Also in our local files an output directory containing all the answers has been created

## Summary

We have successfully written a simple *MapReduce* application using *PySpark* and launched it on a *Spark* cluster which contained 1 master node and 2 worker nodes!