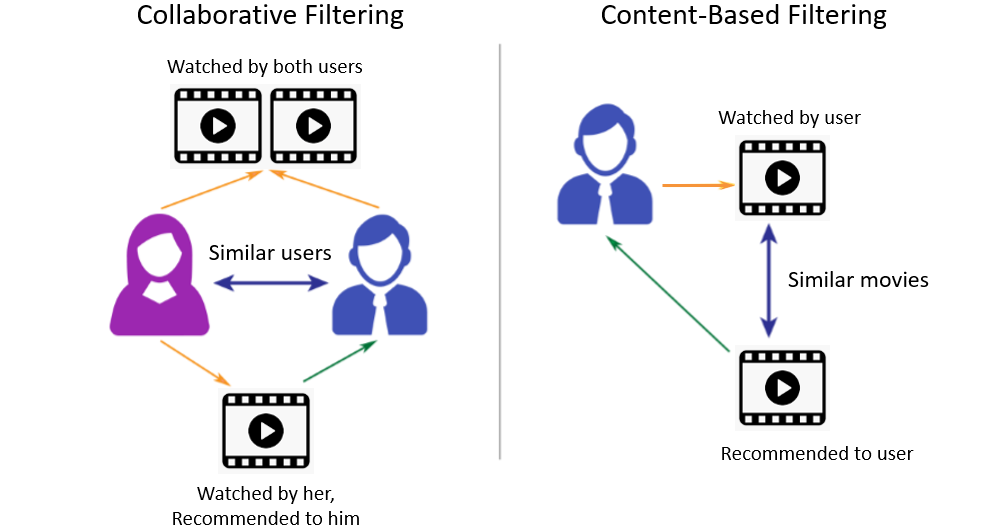
**Movie Recommendations System (Spark, SQL with Python)**



Thanks to the advancement of technology in the film industry in recent years, more and more people are enjoying their favorite flicks at the comfort of their sofas. Therefore, it is no surprise to see the gargantuan fights for supremacy between different movies streaming services like Netflix, Hulu, Disney+, etc. across news headlines (Alexander, 2020). Despite having distinct marketing strategies and original contents, all of them share these websites share one common thing: ”movie recommendation system”.

Movies are suggested to the viewers through 2 methods: Collaborative filtering or Content-based filtering. Content based filtering made their predictions based on the genes of your watch history, if you like ‘Friends’ or ‘Zootopia’ then you will see ‘Shazam, which is pure ‘comedy’, in your recommendation list. On the other hand, Collaborative Filtering system gets it prediction based on similar users’ history. Movies streaming services use both simultaneously to get the best results. (Kirzhner, 2018)

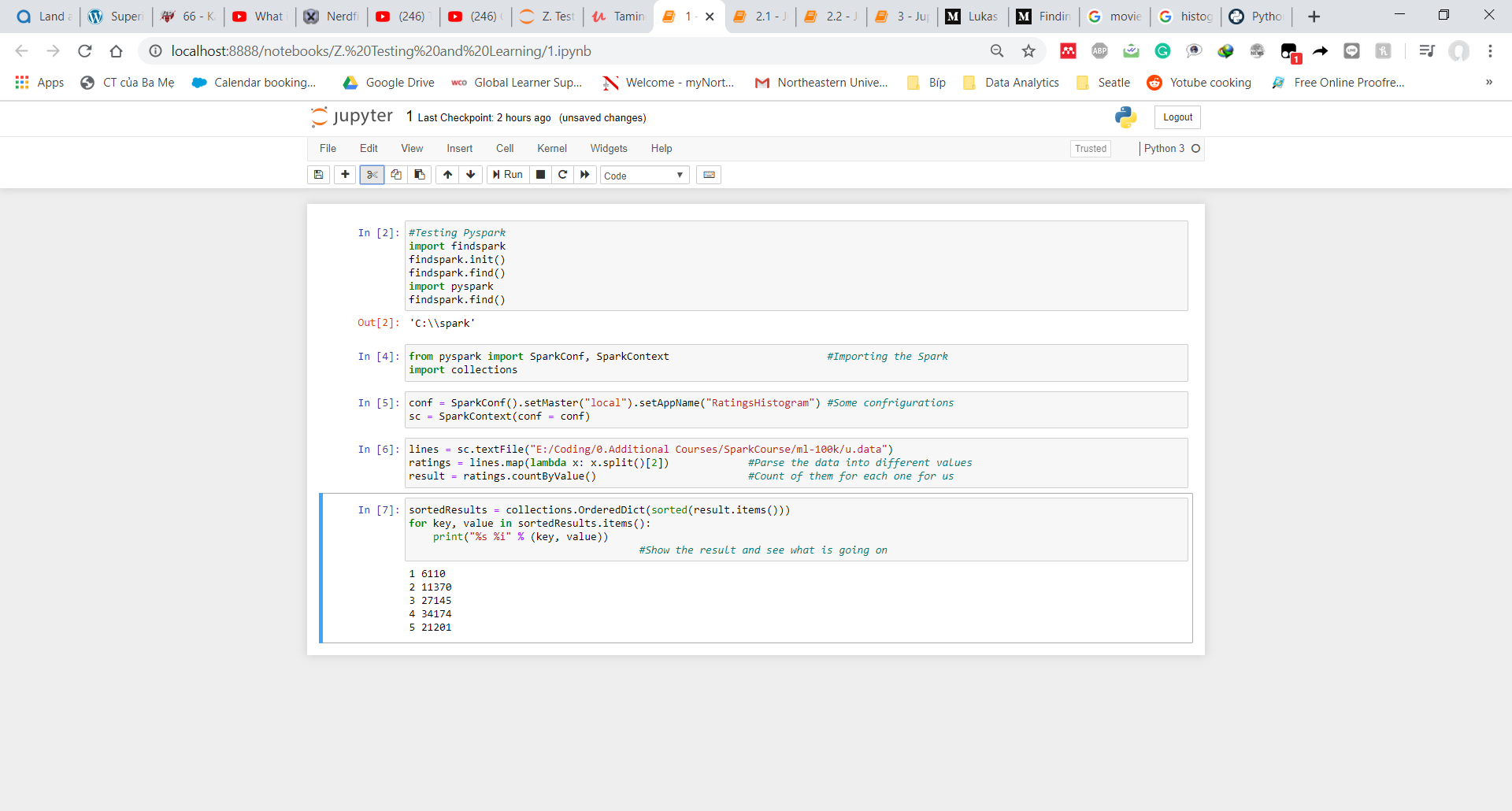


In this project, we will explore the Collaborative Filtering approach to get the movie recommendations for the StarWar (1997) movie using Spark code in Python. The recommendations are generated by the watched history of those who watched ‘StarWar (1997) while reviewed other movies. If they rate a ‘StarWar (1997)’ highly, chances are, they also think the same thing with other similar flicks. The project uses three separate datasets:

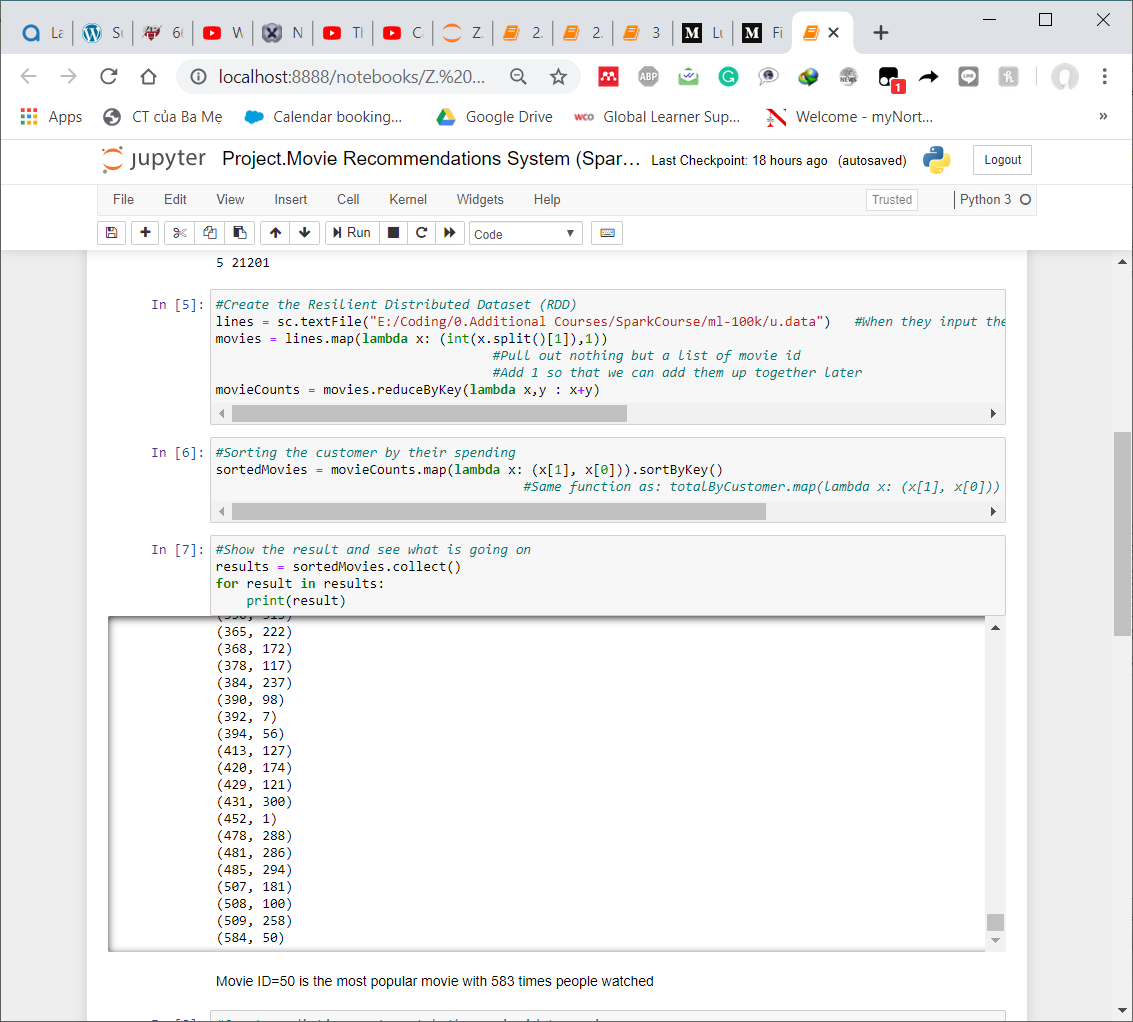
* ‘The u.ITEM file contains: ‘MovieID’, ‘Movie Name’
* The u.DATA file contains: ‘UserID’, ‘MovieID’, ‘‘MovieRating’,’Time Stamp’ of 100,000 movies ratings. The dataset was from 1998
* The ‘ratings’ file contains similar features like the ‘u.DATA’ file but for 1 million movie ratings. The dataset was from 2003

**I/ Exploratory Data Analysis (EDA)**

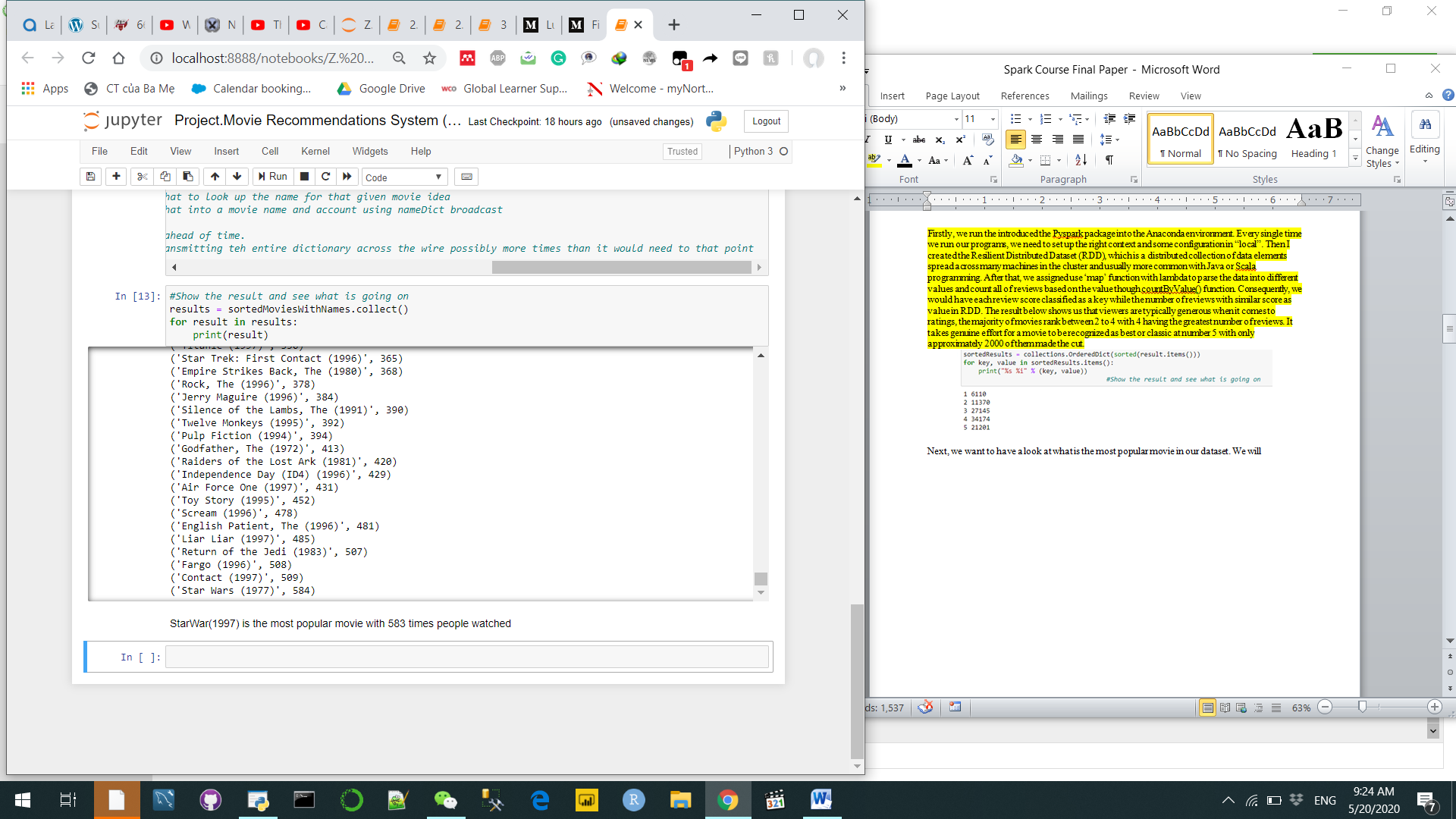
Firstly, we run the introduced the Pyspark package into the Anaconda environment. Every single time we run our programs, we need to set up the right context and some configuration in “local”. Then I created the Resilient Distributed Dataset (RDD), which is a distributed collection of data elements spread across many machines in the cluster and usually more common with Java or Scala programming. After that, we assigned use ‘map’ function with lambda to parse the data into different values and count all of reviews based on the value though countByValue() function. Consequently, we would have each review score classified as a key while the number of reviews with similar score as value in RDD. The result below shows us that viewers are typically generous when it comes to ratings, the majority of movies rank between 2 to 4 with 4 having the greatest number of reviews. It takes genuine effort for a movie to be recognized as best or classic at number 5 with only approximately 2000 of them made the cut.



Next, We will create ‘movies’ variable using the map() function and lambda method to pull out nothing but a list of ‘MovieID’ as key and assign each line’s value as ‘1’ so that we can add them up together later. Then, we count the number that a movie was review by adding up all the lines that share the same ‘MovieID’ key through the reduceByKey() function. After that, we switched the place between the key and the value so that the ‘Number of occurrences’ became keys while the ‘MovieID’ became value. With sortByKey(), we found out ‘50’ is the most popular movie with 584 reviews.



To check what movie it is, we built a dictionary to match the ‘MovieID’ with the ‘MovieNames’. Then we finally found that it was ‘Star Wars (1997)’ all along



**II/ Movie Recommendations and potential model optimization methods**

In order to recreate the movie recommendations system, we will have to follow three steps. Firstly, wefind every pair of movies that were watched by the same person. Then, we measure the similarity of their ratings across all users who watched both. Finally, we sort by movie, then by similarity strength

**Approach by code:**

* Map input ratings to (userID,(movieID,rating))
* Find every movie pair rated to the same user

This can be done with a "self-join" operation

At this point we have (userID,((movieID1,rating1),(movieID2,rating2)))

- Filter out duplicate pairs

- Make the movie pairs the key

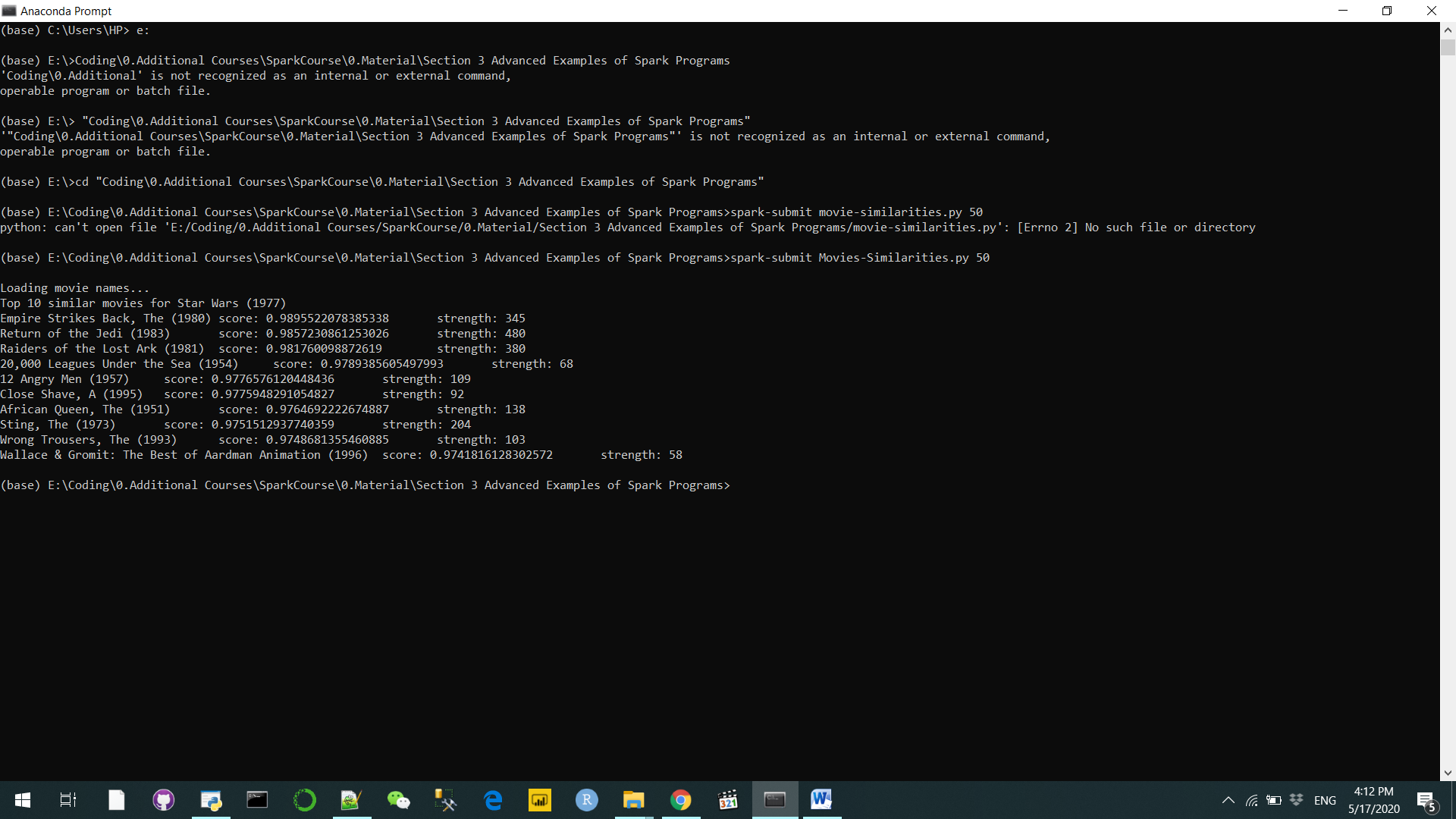
map to ((movieID1,movieID2), (rating1,rating2) /// movieID: keys, rating: values

- groupByKey() to get every rating pair found for each movie pair

- Compute similarity between ratings for each movie in the pair

- Sort, save & display results

It is noted that the model we are using implemented the ‘Coside metrics’. We also set the threshold for the minimum co-raters or minimum scores as 0.95 and the Occurrence threshold as 50 . Unlike the other project, this cannot be done with the script on Jupyter Notebook but rather Anaconda prompt. In the Anaconda prompt from your start menu and run the command ' spark-submit Movies-Similarities.py 50'. Make sure to save the file in .py rather than .ipynb format and run your code on the right drive so that it will run smoothly. Once done, we have the final recommendations below:



‘Score’ indicates how accurate your prediction is and ‘Strength’ symbolizes the number of similar users who watched ‘Star Wars (1977)’ also watched that movie. ‘Score’ has more credibility in this report so that is why we have ‘Empire Strike Back’ being recommended before ‘Return of the Jedi’ even though the latter has lower strength. However, we need to make sure to moderate the ‘Strength’ threshold well since we have some unrelated recommendations like ‘20,000 Leagues Under the Sea’, ‘Close Shave, A ’ ‘Wallace & Gromit’ with only fewer than 100 similar reviews.

There are five ways to improve the results that future researchers to improve the model:

* Discard all the bad ratings like ‘1’ and keep only the good recommendations
* Try different similarity metrics instead of the ‘Coside metrics’ such as Pearson Correlation Coefficient, Jacquard Coefficient, Conditional Probability
* Invent a new similarity metric that takes the number of co-raters into account
* Merge the ‘Content filtering approach’ with the’ Collaborative filtering’ by using the genre dataset to realistically stimulate a Movie recommendation system
* Adjust the thresholds that we set above

In this instance, we would look at the besides the top 3, what are the other 3 recommendations that the system will recommend we modified our thresholds. The reasons why we ignored the first three movies were because they all have ‘Score’ rates above 300, rendering our comparison useless if we include them.



From the able above, it is clear the last threshold scores combination is the best since it includes all beloved classics that are similar to the ‘Star War 1977’ in term of genres: adventure, action and sci-fi

**III/ Movie Recommendations on the Cloud Cluster (Amazon Web Service-EC2)**

In this part w**e throw in a million movie ratings instead of just 100.000**, using EC2 in AWS to run Spark on the Cloud Cluster.

**a) Introduction to AWS Elastic MapReduce and PartitionBy()**

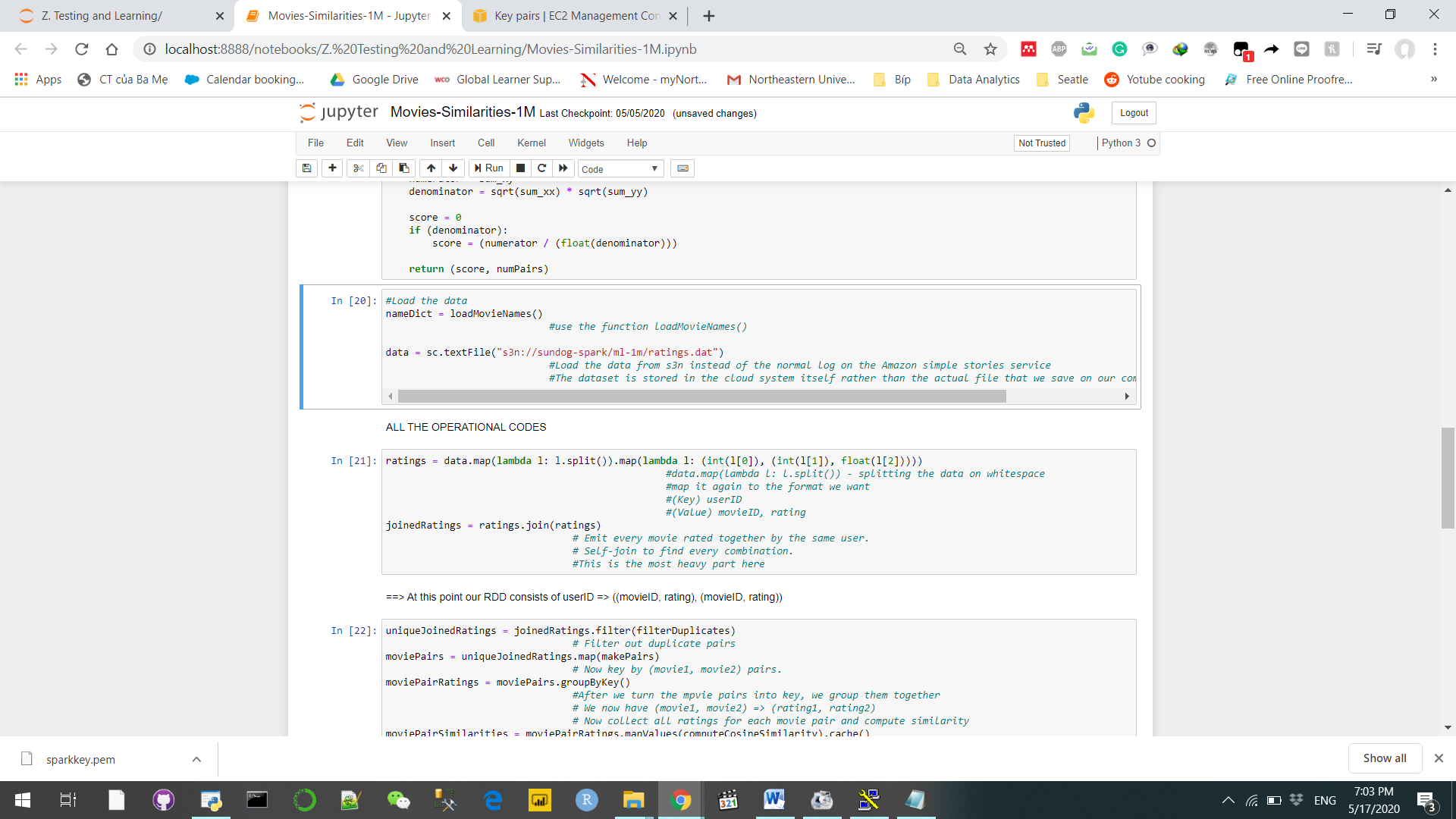
**AWS Elastic MapReduce:**

* Very quick and easy way to rent time on a cluster of your own
* Sets up a default spark configuration for you on top of Hadoop’s Yarn cluster manager
* Spark also has a built in standalone cluster manager and scripts to set up its own EC2-based cluster (Barr, 2009)
* Spark on EMR isn’t really expensive but it is not cheap either
* Make sure things run locally on a subset of your data first

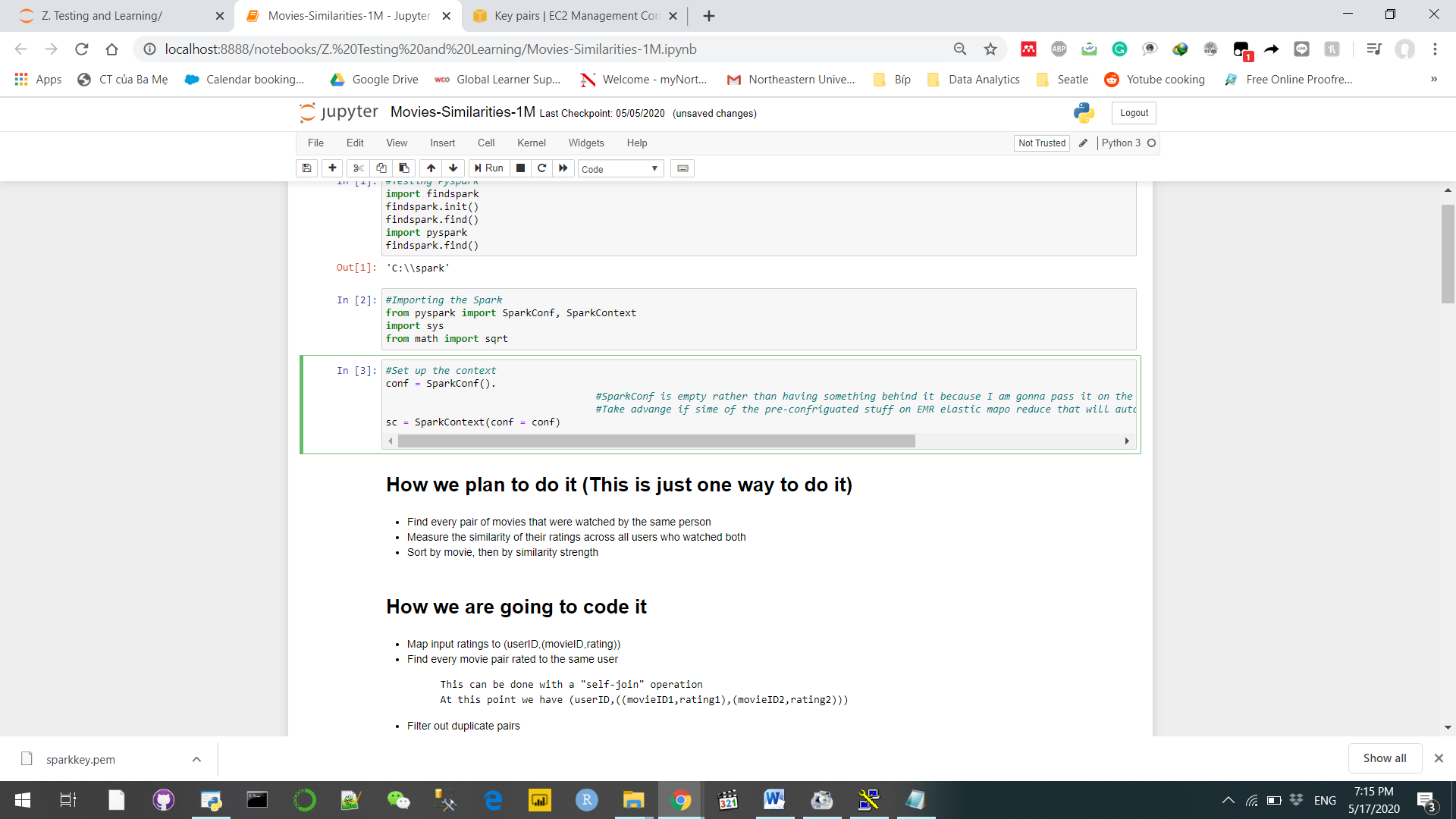
**PartitionBy():** is a method on an RDD that you can use to say “Hey I’m gonna run some large operation but I don’t have enough computer resourses to actually split this up into many different executors, many different runs” This basically tells me how many pieces I want to break this job up into . It is very important to determine how many partitions are enough

* Too few and we wont take full advantage of your cluster
* Too many results in too mych overhead from shuffling data
* At least as many partititions as you have cores or executors that fit within your available memory
* ParticionBy(100) across 10 computers is usually a reasonable place to start for large operations

**b) Process and review:**



1/ We will use the u.DATA’ file but for 1 million movie ratings. We load the data from s3n instead of the normal log on the Amazon simple stories service. The dataset is stored in the cloud system itself rather than the actual file that we save on our computer. (Sundog-Spark). SparkConf is empty rather than having something behind it because I am gonna pass it on the command line. Take advantage of some of the Pre-congregated stuff on EMR elastic Map Reduce that will automatically tell Spark to run on top of Hadoop Yarn

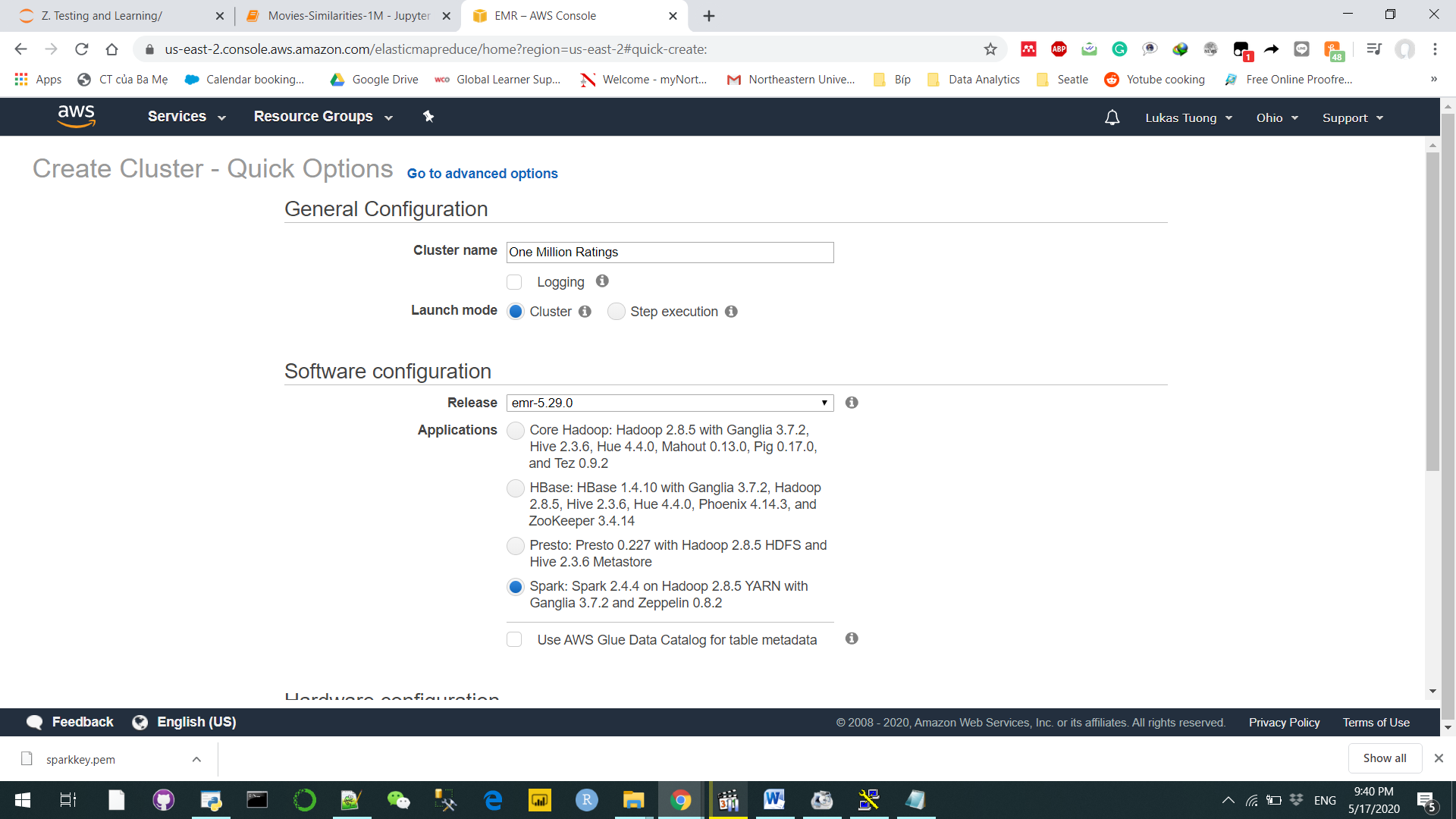


2/ Repeat the code that we created in the second part.

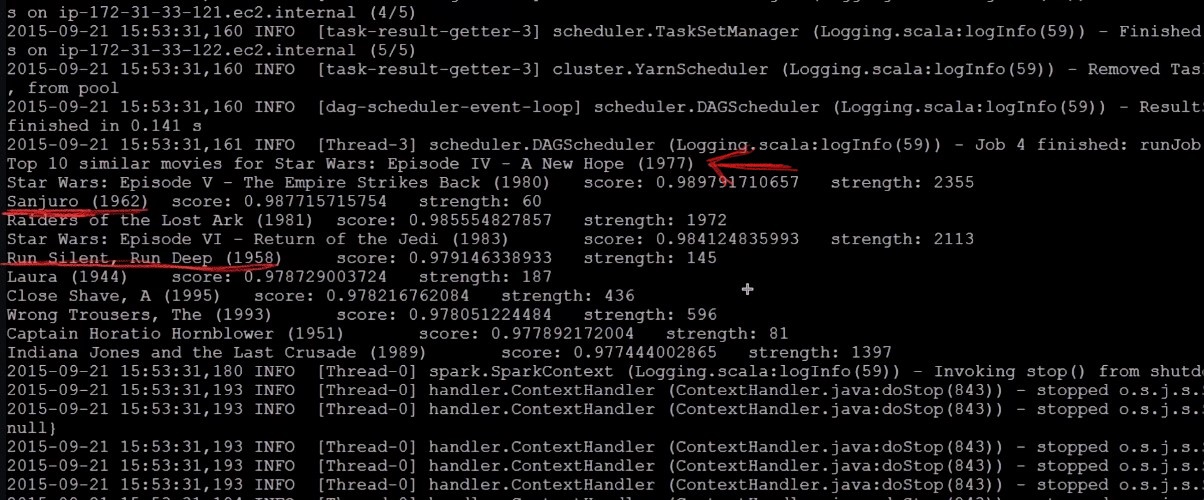
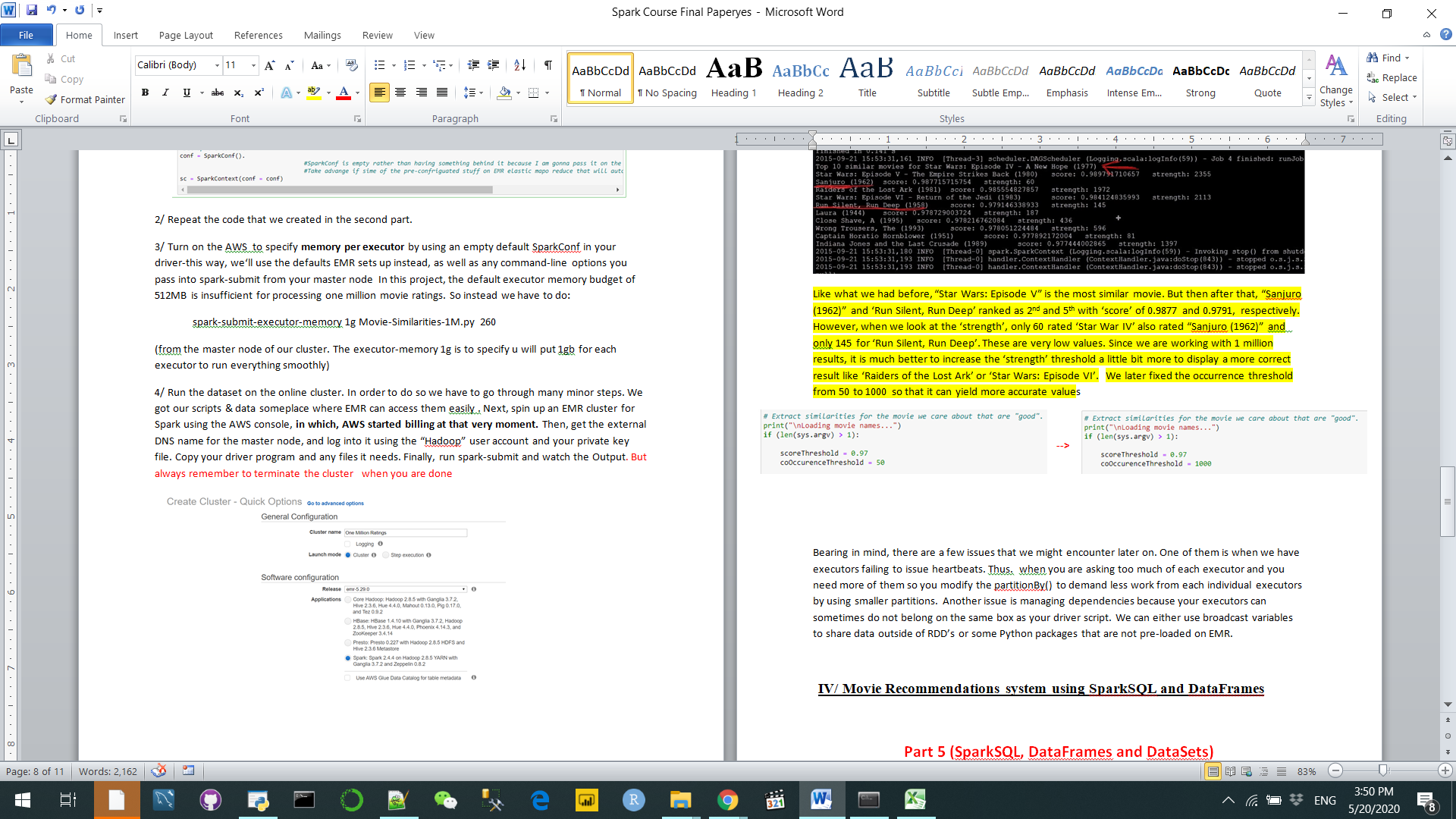
3/ Turn on the AWS to specify **memory per executor** by using an empty default SparkConf in your driver-this way, we‘ll use the defaults EMR sets up instead, as well as any command-line options you pass into spark-submit from your master node In this project, the default executor memory budget of 512MB is insufficient for processing one million movie ratings. So instead we have to do:

spark-submit-executor-memory 1g Movie-Similarities-1M.py 260

(from the master node of our cluster. The executor-memory 1g is to specify u will put 1gb for each executor to run everything smoothly)



4/ Run the dataset on the online cluster. In order to do so we have to go through many minor steps. We got our scripts & data someplace where EMR can access them easily . Next, spin up an EMR cluster for Spark using the AWS console, **in which, AWS started billing at that very moment.** Then, get the external DNS name for the master node, and log into it using the “Hadoop” user account and your private key file. Copy your driver program and any files it needs. Finally, run spark-submit and watch the Output. But always remember to terminate the cluster when you are done



Like what we had before, “Star Wars: Episode V” is the most similar movie. But then after that, “Sanjuro (1962)” and ‘Run Silent, Run Deep’ ranked as 2nd and 5th with ‘score’ of 0.9877 and 0.9791, respectively. However, when we look at the ‘strength’, only 60 rated ‘Star War IV’ also rated “Sanjuro (1962)” and only 145 for ‘Run Silent, Run Deep’. These are very low values. Since we are working with 1 million results, it is much better to increase the ‘strength’ threshold a little bit more to display a more correct result like ‘Raiders of the Lost Ark’ or ‘Star Wars: Episode VI’. We later fixed the occurrence threshold from 50 to 1000 so that it can yield more accurate values

Bearing in mind, there are a few issues that we might encounter later on. One of them is when we have executors failing to issue heartbeats. Thus,when you are asking too much of each executor and you need more of them so you modify the partitionBy() to demand less work from each individual executors by using smaller partitions. Another issue is managing dependencies because your executors can sometimes do not belong on the same box as your driver script. We can either use broadcast variables to share data outside of RDD’s or some Python packages that are not pre-loaded on EMR.

**IV/ Movie Recommendations system using Spark’s Machine Learning Library (MLlib)**

**a) MLlib abilities**

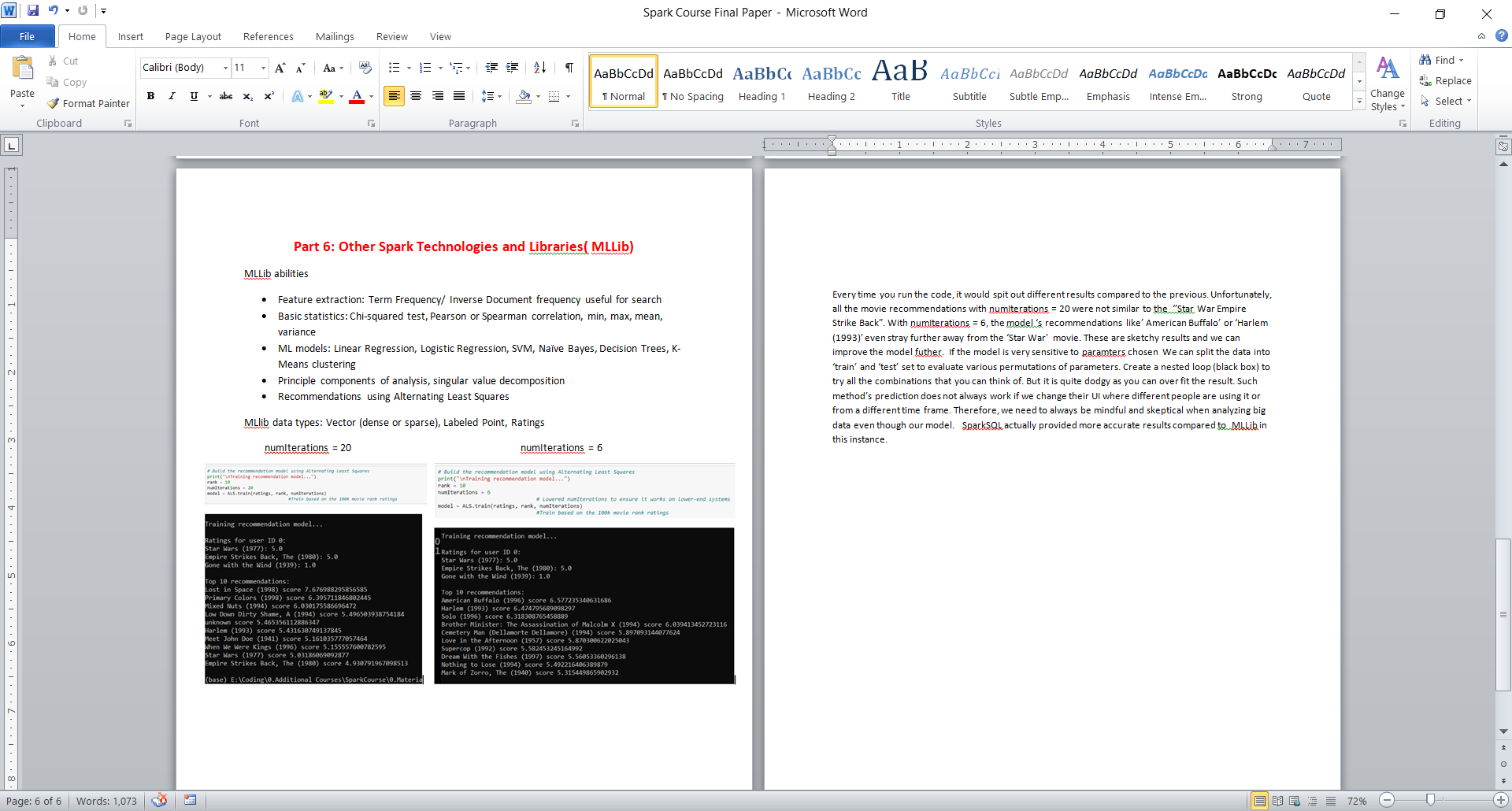
* Feature extraction: Term Frequency/ Inverse Document frequency useful for search
* Basic statistics: Chi-squared test, Pearson or Spearman correlation, min, max, mean, variance
* ML models: Linear Regression, Logistic Regression, SVM, Naïve Bayes, Decision Trees, K-Means clustering (McDonald, 2019)
* Principle components of analysis, singular value decomposition
* Recommendations using Alternating Least Squares

MLlib data has three types: Vector (dense or sparse), Labeled Point, Ratings

**b) Process and review:**



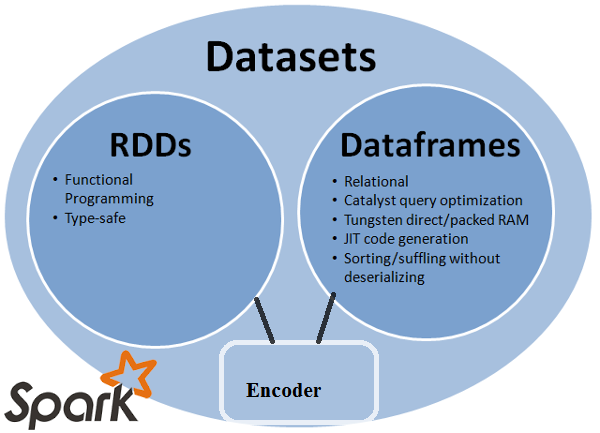
We first added 3 more fake reviews like above to the u.Data to test our model, created the configuration and checkpoints. Then, we loaded the data and trained the model



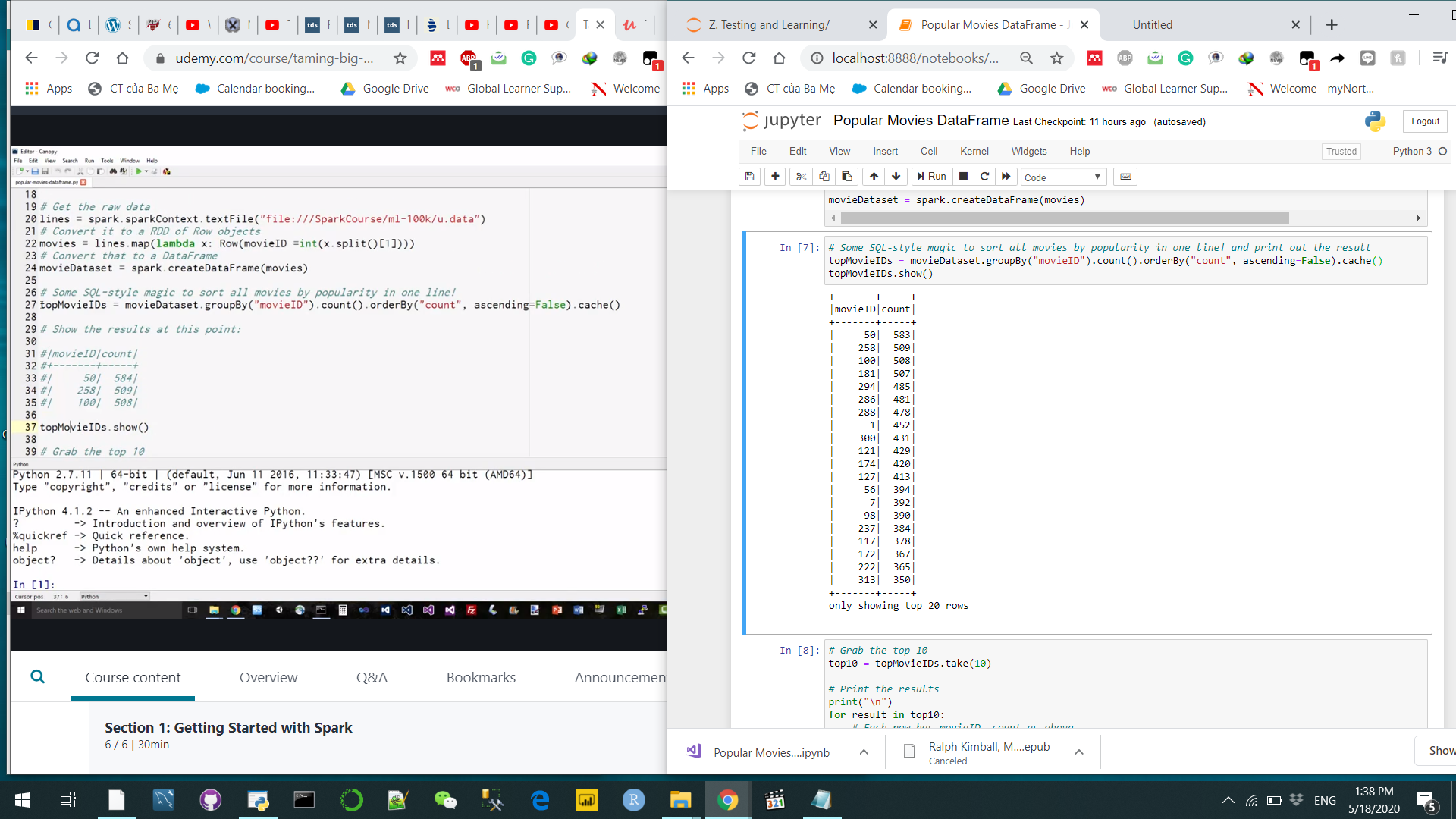
Every time you run the code, it would spit out different results compared to the previous. Unfortunately, all the movie recommendations with numIterations = 20 were not similar to the ‘’Star War Empire Strike Back”. With numIterations = 6, the model ‘s recommendations like’ American Buffalo’ or ‘Harlem (1993)’ even stray further away from the ‘Star War’ movie. These are sketchy results and we can improve the model futher. If the model is very sensitive to paramters chosen We can split the data into ‘train’ and ‘test’ set to evaluate various permutations of parameters. Create a nested loop (black box) to try all the combinations that you can think of. But it is quite dodgy as you can over fit the result. Such method’s prediction does not always work if we change their UI where different people are using it or from a different time frame. Therefore, we need to always be mindful and skeptical when analyzing big data even though our model. normal Spark actually provided more accurate results compared to MLLib in this instance.

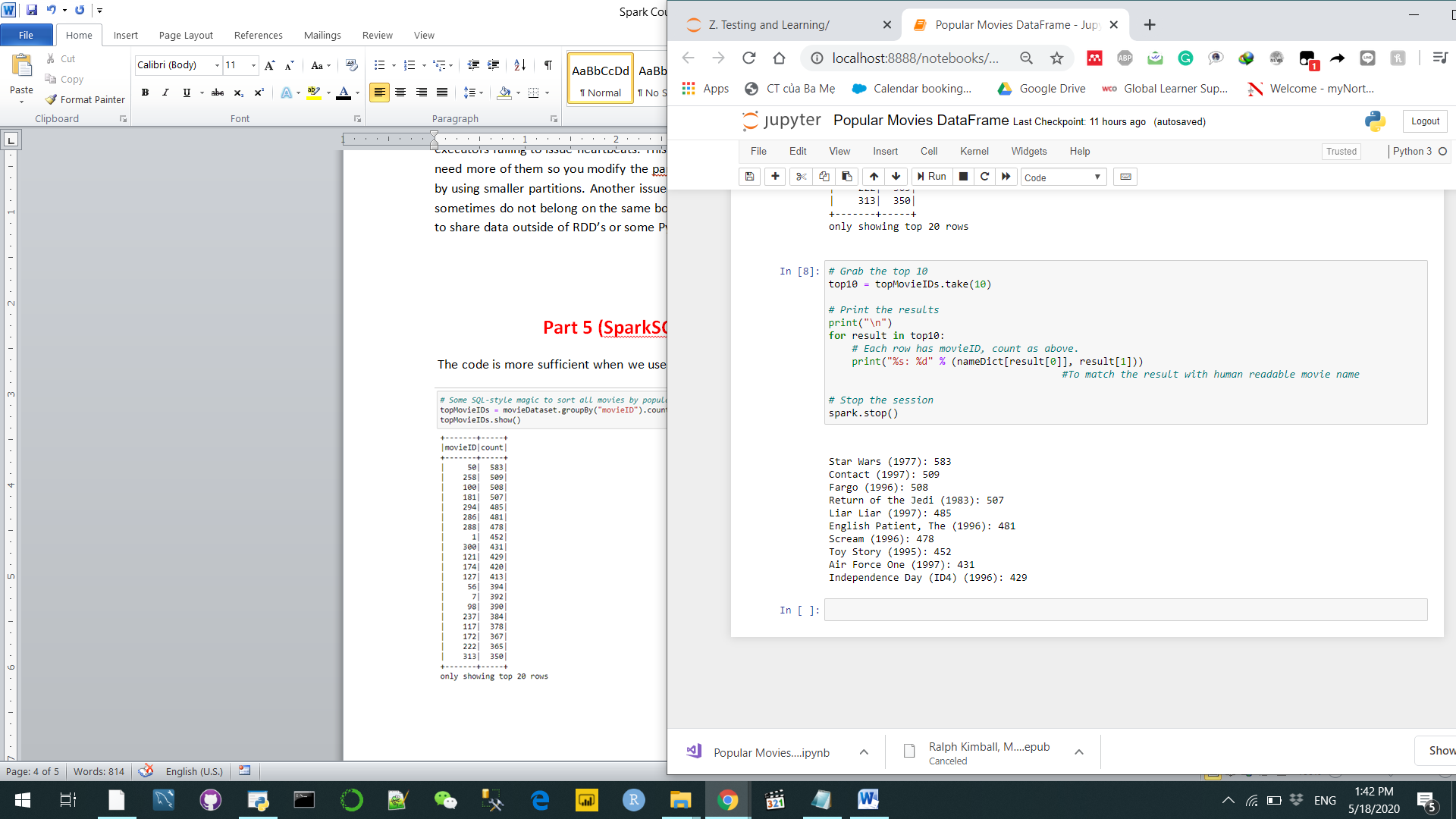
**V/ Simplify the code with SparkSQL and DataFrames**

In this part of the project, we will see have a look of how SparksSQL can be help us to simplify our code with DataFrames compared to RDD. Dataframe belongs to the Dataset umbrella, DataFrame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column. (Geeksforgeeks, 2019) It is considered to be the future of Spark thanks to its distanced advantages compared to the RDD.



Instead of the big chunks of codes like earlier, the code is more sufficient when we used the DataFrame rather than the RDD format with SQL. All the actions can be rendered into one single code line as well can see below. We could simultaneously group the data according their ID, then count them and then sort them by number of reviews for a movies in ascending order. We achieved the same EDA analysis in a faster pace compared to the part I with ‘Star War(1997)’ being the most popular movie.





**VI/ Conclusion**

This project is a good opportunity for us to understand how Spark was implemented in building an effective Movie Recommendation system in the according to the Collaborating Filtering method. It also proves that machine learning is not always about using the most advanced technology but rather how well we actually understand issue. Evdinely shown through the triumphant of simple Spark code over complicated MLlib

**Reference & Sources:**

Geeksforgeeks. (2019). Python | Pandas DataFrame. Retrieved from <https://www.geeksforgeeks.org/python-pandas-dataframe/>

Alexander, J. (2020). The streaming wars are finally beginning, but it’s more of a polite quarrel than an all-out war. Retrieved from <https://www.theverge.com/2020/2/6/21126156/streaming-wars-disney-plus-netflix-wall-street-subscribers-hbo-max-peacock>

McDonald, C. (2019). Apache Spark Machine Learning Tutorial | MapR. Retrieved from <https://mapr.com/blog/apache-spark-machine-learning-tutorial/>

Barr, J. (2009, April 2). Announcing Amazon Elastic MapReduce. Retrieved from <https://aws.amazon.com/blogs/aws/announcing-amazon-elastic-mapreduce/>

Kirzhner, E. (2018). Machine Learning. Explanation of Collaborative Filtering vs Content Based Filtering. Retrieved from <https://codeburst.io/explanation-of-recommender-systems-in-information-retrieval-13077e1d916c>

Movie Review Dataset: <https://grouplens.org/datasets/movielens/>