

COMP5439 Assignment 2

Implementation of Spark APIs to Analyse Twitter User Data

Christian Sullivan

1 Introduction

This implementation will demonstrate the use of Spark SQL and Spark ML packages called through the PySpark API. The implementation serves two key purposes:

1. To identify the top 5 users with a similar interest to a given user id using word tokenization.
2. To apply a collaborative filtering algorithm to identify and recommend other user ids to each user.

This report will analyse and discuss the implementation of each workload to describe implementations, optimisations and to analyse the performance of each workload using both local and an implementation in Amazon's EMR.

2 Workload 1: Measuring the Cosine Distance of a given user to other users

2.1 Design

For this workload the data was initially processed and vectorised before two separate word tokenization methods were implemented and compared: Term Frequency and Count Vectorizer.

2.1.1 Data Processing

The purpose of this section was to process and clean the data to be then tokenized by creating a documentation representation column with the retweet ids and reply ids of each user. This was done by grouping the data by *user_id* and running a *collect_list* function on the *replyto_id* and *retweet_id* columns which would combine each value into column

of lists, using `concat_ws` to transform the lists into strings separated by a space. Prior to combining the retweets and replies columns each blank value was set to null to prevent issues when they were concatenated into a new column: `agg_tweet_respond` which contains the required document representation of for each `user_id`. This is visualised below:

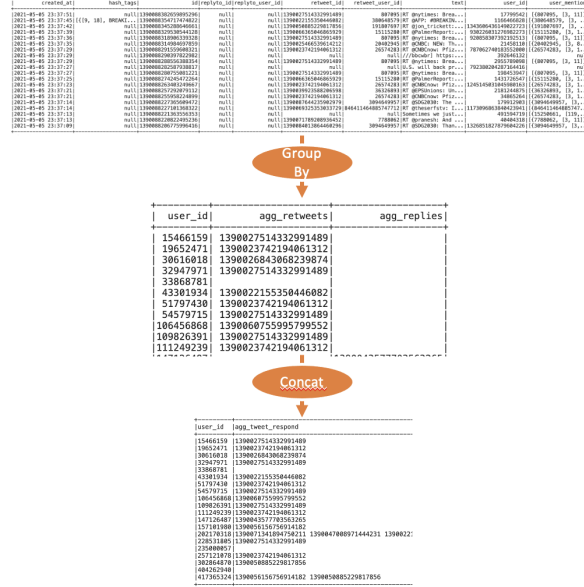


Figure 1: WL1 Data Processing DAG

2.1.2 Tokenizing and Calculating Cosine Similarity

With the data now cleaned the data was vectorized using the Tokenizer function to ensure that it could be processed using the Term Frequency and Count Vectorizer methods. The Term Frequency and Count Vectorizer methods were applied to the tokenized data with the output appearing as below:

TF Method		Count Vectorizer Method	
user_id	tf	user_id	features
15466159	(262144, [23124], [1,0])	15466159	(656, [0], [1,0])
19652471	(262144, [204285], [1,0])	19652471	(656, [1], [1,0])
30616018	(262144, [158844], [1,0])	30616018	(656, [114], [1,0])
32947971	(262144, [23124], [1,0])	32947971	(656, [0], [1,0])
33868781	(262144, [249180], [1,0])	33868781	(656, [2], [1,0])
43381934	(262144, [178090], [1,0])	43381934	(656, [3], [1,0])
51797430	(262144, [204285], [1,0])	51797430	(656, [1], [1,0])
54579715	(262144, [23124], [1,0])	54579715	(656, [0], [1,0])
106456868	(262144, [95155], [1,0])	106456868	(656, [14], [1,0])
109826391	(262144, [23124], [1,0])	109826391	(656, [0], [1,0])
111249239	(262144, [204285], [1,0])	111249239	(656, [1], [1,0])
147126487	(262144, [247891], [1,0])	147126487	(656, [480], [1,0])
157101980	(262144, [178090], [1,0])	157101980	(656, [26], [1,0])
202170318	(262144, [78232, 127139, 151508, 178090, 232689], [1,0,1,0,1,0,1,0,1,0])	202170318	(656, [3,16,25,87,100], [1,0,1,0,1,0,1,0,1,0])
228531805	(262144, [23124], [1,0])	228531805	(656, [0], [1,0])
235080857	(262144, [249180], [1,0])	235080857	(656, [2], [1,0])
257121878	(262144, [204285], [1,0])	257121878	(656, [1], [1,0])
302864870	(262144, [101572], [1,0])	302864870	(656, [0], [1,0])
404262940	(262144, [249180], [1,0])	404262940	(656, [2], [1,0])
417365324	(262144, [178090, 181572], [1,0,1,0])	417365324	(656, [8,26], [1,0,1,0])

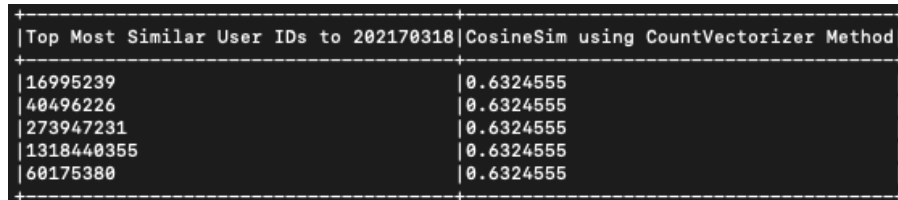
Figure 2: Output of TF and Count Vectorizer Methods

The Term Frequency and Count Vectorizer were each filtered to get the features of the selected user to calculate the cosine similarity against each other user. A cosine similarity function was calculated using the following formula and code:

$$similarity(A, B) = \frac{A \cdot B}{||A|| * ||B||} \quad (1)$$

```
def cos_sim(a,b=compare_vector):
    return float(a.dot(b) / (a.norm(2) * b.norm(2)))
cos_function = udf(cos_sim, FloatType())
```

Finally, the cosine similarity between the selected user and each other user was calculated, sorted and the top 5 were printed as captured below:



Top Most Similar User IDs to 202170318	CosineSim using CountVectorizer Method
16995239	0.6324555
40496226	0.6324555
273947231	0.6324555
1318440355	0.6324555
60175380	0.6324555

Figure 3: Output of Cosine Similarities for User 202170318 using CountVectorizer Method

2.1.3 Optimisations

To ensure optimal performance, caching was added in a number of steps to minimise the volume of repeat calculations that needed to be applied to the same dataset. In this workload, the data was cached following the initial group by on *user_id* to avoid any requirement to retransform the data in the subsequent transforms. Further caching operations were also added after the Term Frequency and Count Vectorizer methods were applied and after the cached dataframes were used, they were unpersisted to reduce memory expenditure. Additionally, when initialising the Spark session, the number of partitions was set to 100 instead of 200 given the data is only 10,000 rows which nullifies the need for so many partitions.

2.2 Performance Analysis

It seems that the Count Vectorizer method was slightly more optimal in terms of performance time than the Term Frequency method - this was mainly due to the process of sorting and displaying the results rather than applying the methods where a Shuffle Read of 164.8KiB was applied to the TF method. This is articulated in the below figure:



Figure 4: WL1 Comparison of Show Step TF vs. CV

When investigating the runtimes of the algorithms themselves it seems that the Term Frequency method was more optimal and involved far less parallelism than Count Vectorizer. Term Frequency was completed across a single task whereas the Count Vectorizer method had to be applied across 100 tasks and required an additional reduceByKey and map step which wasn't experienced in Term Frequency. This is likely due to the TF option leveraging hashing to generate word tokens (Dhamija, 2019). The single operation of the TF algorithm was only 1 second while the multiple operations across the two stages for CV took 2.4 seconds. this is indicated below:

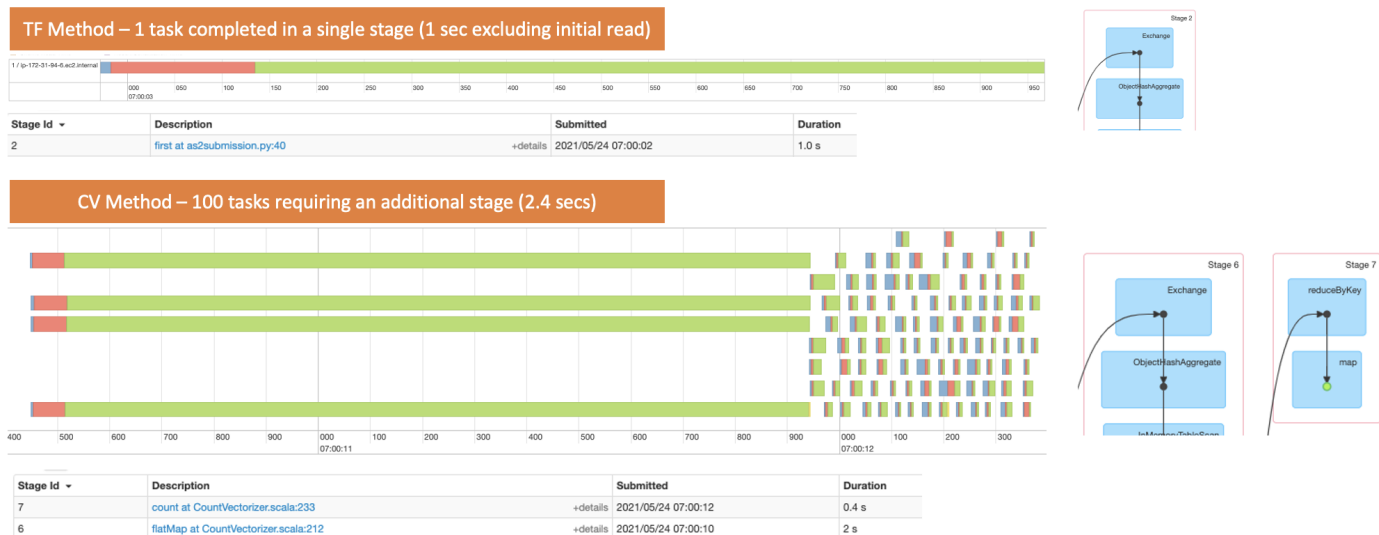


Figure 5: WL1 Comparison of Algorithms TF vs. CV

3 Workload 2: Applying a collaborative filtering algorithm to identify and recommend other user ids to each user

3.1 Design

For this workload the data was processed by extracting the user id for each mention user before determining the top 5 user recommendations for each user. This algorithm applies collaborative filtering with alternating least squares to develop a recommendation engine, recognising similarities between users based on other users they mention in their tweets (Aljunid & Manjaiah, 2019).

3.1.1 Data Processing

First the *id* field from the *user_mentions* column was extracted for each user to create a new column consisting purely of the user ids of each mentioned user. Then an explode function was applied to this column to separate each mentioned user into a separate row of data. The data was then grouped by *user_id* and *mentioned_users* to create a count of each time a user was mentioned. Finally, each recommendation per user was expanded to an individual column with 5 recommend user columns per *user_id*. This process is mapped out below:

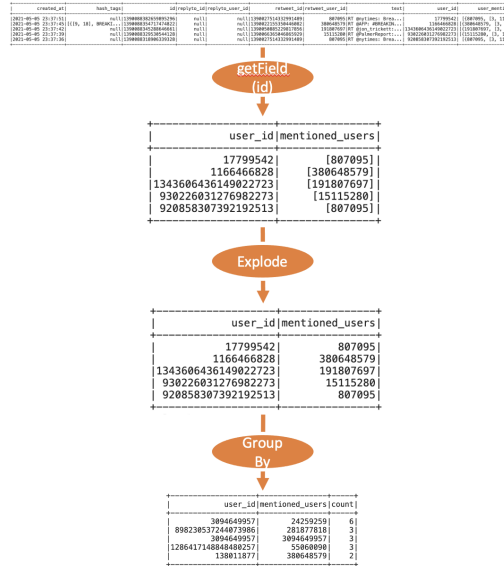


Figure 6: WL2 Data Processing Overview

3.1.2 Modelling the Data

Given the ALS model requires numeric data, a string indexer was applied both to the *user_id* and *mentioned_users* to create a numerical input. With this completed the ALS algorithm was applied and the top 5 user recommendations were supplied for each user. To ensure the results of this were meaningful, the results were transformed back by taking the labels from the *StringIndexer* function and converting them back to the original IDs as below:

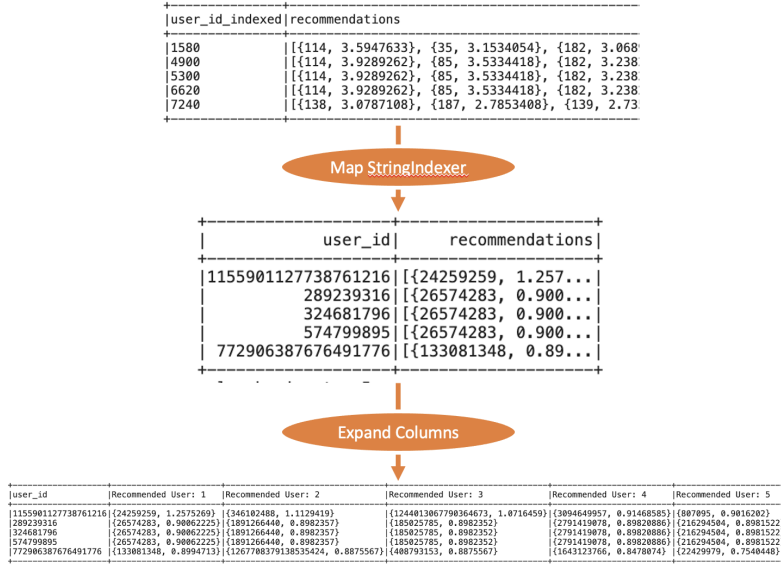


Figure 7: WL2 Modelling Overview

3.1.3 Optimisations

As with the above section, the number of tasks for this job was set to 100. This workload was split into two key functions: the first to process the data and the second to apply the ALS algorithm. After the initial preprocessing step, caching was applied to reduce the need for further operations. In the modelling step, caching was applied on the dataframe containing initial recommendations for each user. In applying the algorithm, observations by Aljunid and Manjaiah, 2019 found that optimal performances were observed with 10 max iterations which corresponds to the default in the ALS API, so no adjustments to model parameters were required.

3.2 Performance Analysis

This step accounted for over half of the time spent completing all workloads which was largely attributed to the time spent applying and reversing the *StringIndexer* rather than applying the ALS algorithm itself. This is likely caused by the *loop* to de-index and map the real IDs of each recommend user in the recommendations column. I had attempted

to store this column in memory which may have sped up the process and without the below looping function the *StringIndexer* would be able to more efficiently apply parallel indexation (Wei & JaJa, 2012).

```
recommendations = array(*[struct(
    mu_labels_[col("recommendations")[i]["mentioned_users_indexed"]].alias("userId"),
    col("recommendations")[i]["rating"].alias("rating")
) for i in range(n)])
```

Interestingly as you can see in the below graph of total time spent, there is a large gap between the penultimate tasks ending and the final task beginning. Further research indicated that this could be due to poor partition allocation, however, after experimenting with only 20 partitions and 200 partitions, there was no noticeable shift in the gap between job executions (Manivannan, 2017). After further inspection in the logs it was recommended to adjust the *spark.debug.maxToStringFields*, however, after increasing this limit to as high as 100, there was still no real improvement in the job runtime.

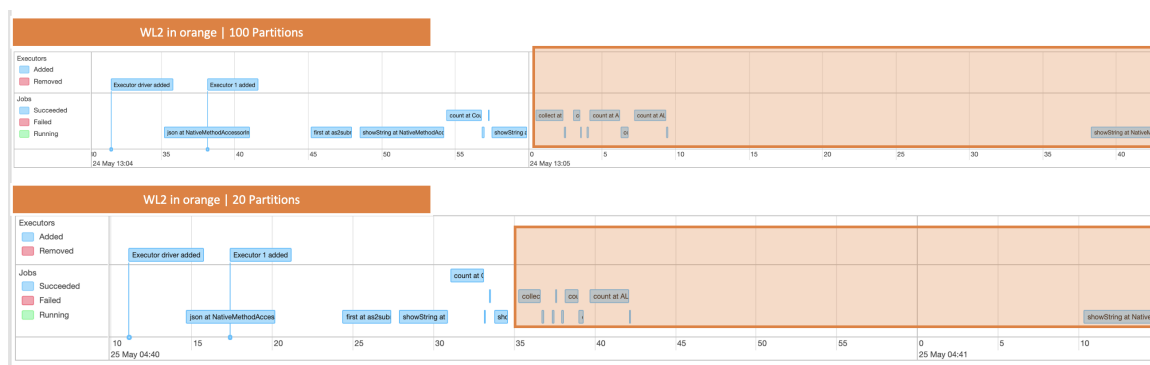


Figure 8: WL2 Partition Comparison

```
21/05/25 04:40:42 INFO Instrumentation: [17593340] training finished
21/05/25 04:40:42 INFO BlockManagerInfo: Removed broadcast_44_piece0 on ip-172-31-86-182.ec2.internal:34061 in memory (size: 237.9 KiB, free: 1027.8 MiB)
21/05/25 04:40:42 INFO BlockManagerInfo: Removed broadcast_44_piece0 on ip-172-31-94-6.ec2.internal:42797 in memory (size: 237.9 KiB, free: 4.8 GiB)
21/05/25 04:40:42 INFO BlockManagerInfo: Removed broadcast_45_piece0 on ip-172-31-86-182.ec2.internal:34061 in memory (size: 237.9 KiB, free: 1028.0 MiB)
21/05/25 04:40:42 INFO BlockManagerInfo: Removed broadcast_45_piece0 on ip-172-31-94-6.ec2.internal:42797 in memory (size: 237.7 KiB, free: 4.8 GiB)
21/05/25 04:41:09 WARN package: Truncated the string representation of a plan since it was too large. This behavior can be adjusted by setting 'spark.sql.debug.maxToStringFields'.
21/05/25 04:41:10 INFO CodeGenerator: Code generated in 23.988485 ms
21/05/25 04:41:10 INFO CodeGenerator: Code generated in 20.030977 ms
21/05/25 04:41:10 INFO CodeGenerator: Code generated in 25.371412 ms
21/05/25 04:41:10 INFO CodeGenerator: Code generated in 16.572975 ms
```

Figure 9: WL2 Logs with gap in timestamp highlighted

The final step to separate out the column of arrays for each user with 5 recommendations columns took place over 5 seconds which was largely due to the *SerializeFromObject* stage below:

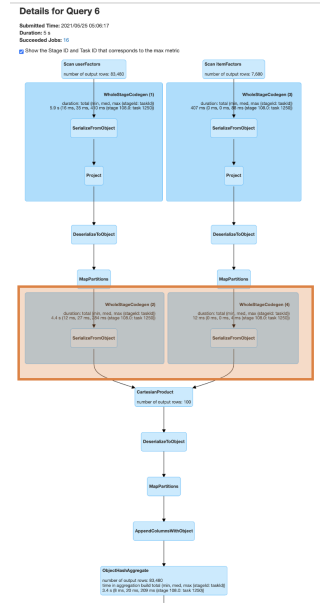


Figure 10: WL2 DAG highlighting the serialize step

4 Conclusion

Ultimately, through various mechanisms, two workloads have been proposed to both measure the Cosine Distance between a given user and all other users, and to create a recommendation engine for users based on the tweet mentions of other recommend users. Further developments could include greater investigation of large breaks between stages in the ALS algorithm.

References

- Aljunid, M. F., & Manjaiah, D. H. (2019). Movie recommender system based on collaborative filtering using apache spark. In V. E. Balas, N. Sharma, & A. Chakrabarti (Eds.), *Data management, analytics and innovation* (pp. 283–295). Springer Singapore.
- Dhamija, V. (2019). Countvectorizer[hashingtf]. <https://towardsdatascience.com/countvectorizer-hashingtf-e66f169e2d4e>
- Manivannan, R. (2017). Apache spark performance tuning – degree of parallelism - dzone performance. <https://dzone.com/articles/apache-spark-performance-tuning-degree-of-parallel>
- Wei, Z., & JaJa, J. (2012). A fast algorithm for constructing inverted files on heterogeneous platforms. *Journal of parallel and distributed computing*, 72(5), 728–738.