THE UNIVERSITY OF SYDNEY

School of Information Technologies
Faculty of Engineering & IT

ASSIGNMENT/PROJECT COVERSHEET - GROUP ASSESSMENT

Unit of Study: COMP5349 Cloud Computing

Assignment name: Text Corpus Analysis

Tutorial time: Thu4-6pm **Tutor name:** Haoyu

DECLARATION

We the undersigned declare that we have read and understood the *University of Sydney Academic Dishonesty and Plagiarism in Coursework Policy*, an, and except where specifically acknowledged, the work contained in this assignment/project is our own work, and has not been copied from other sources or been previously submitted for award or assessment.

We understand that failure to comply with the *Academic Dishonesty and Plagiarism in Coursework Policy* can lead to severe penalties as outlined under Chapter 8 of the *University of Sydney By-Law 1999* (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

We realise that we may be asked to identify those portions of the work contributed by each of us and required to demonstrate our individual knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

Project team members				
Student name	**Student ID**	**Participated**	**Agree to share**	**Signature**
1. Shulei Chen	460332390	[✓ Yes] / No	[✓ Yes] / No	shulei chen
2. Shuhao Liang	500433395	[✓ Yes] / No	[✓ Yes] / No	[illegible]
3.		Yes / No	Yes / No	
4.		Yes / No	Yes / No	
5.		Yes / No	Yes / No	
6.		Yes / No	Yes / No	
7.		Yes / No	Yes / No	
8.		Yes / No	Yes / No	
9.		Yes / No	Yes / No	
10.		Yes / No	Yes / No	

COMP5349 Assignment1 Report

Text Corpus Analysis

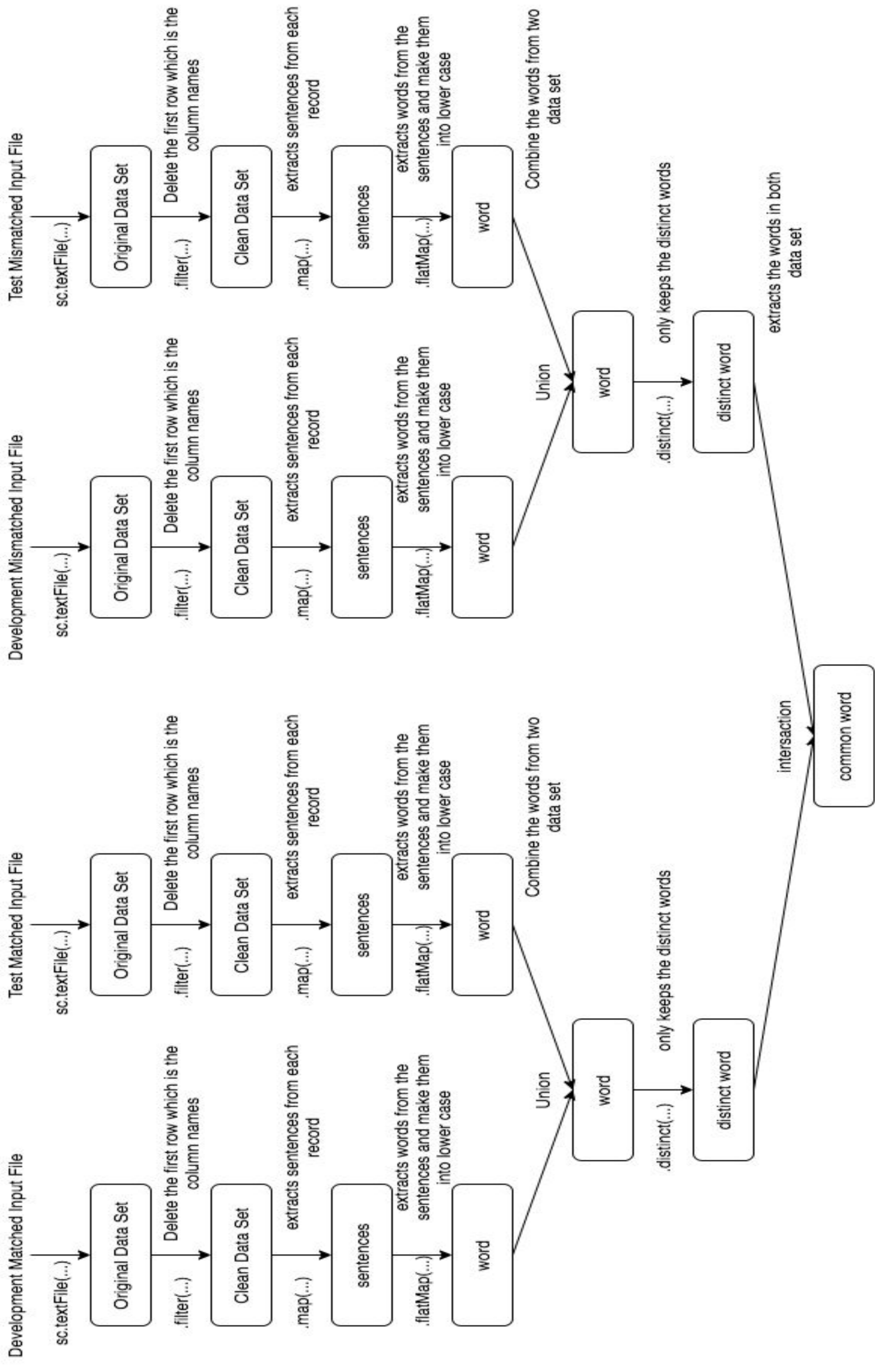
Shuhao Liang, Shulei Chen

Introduction

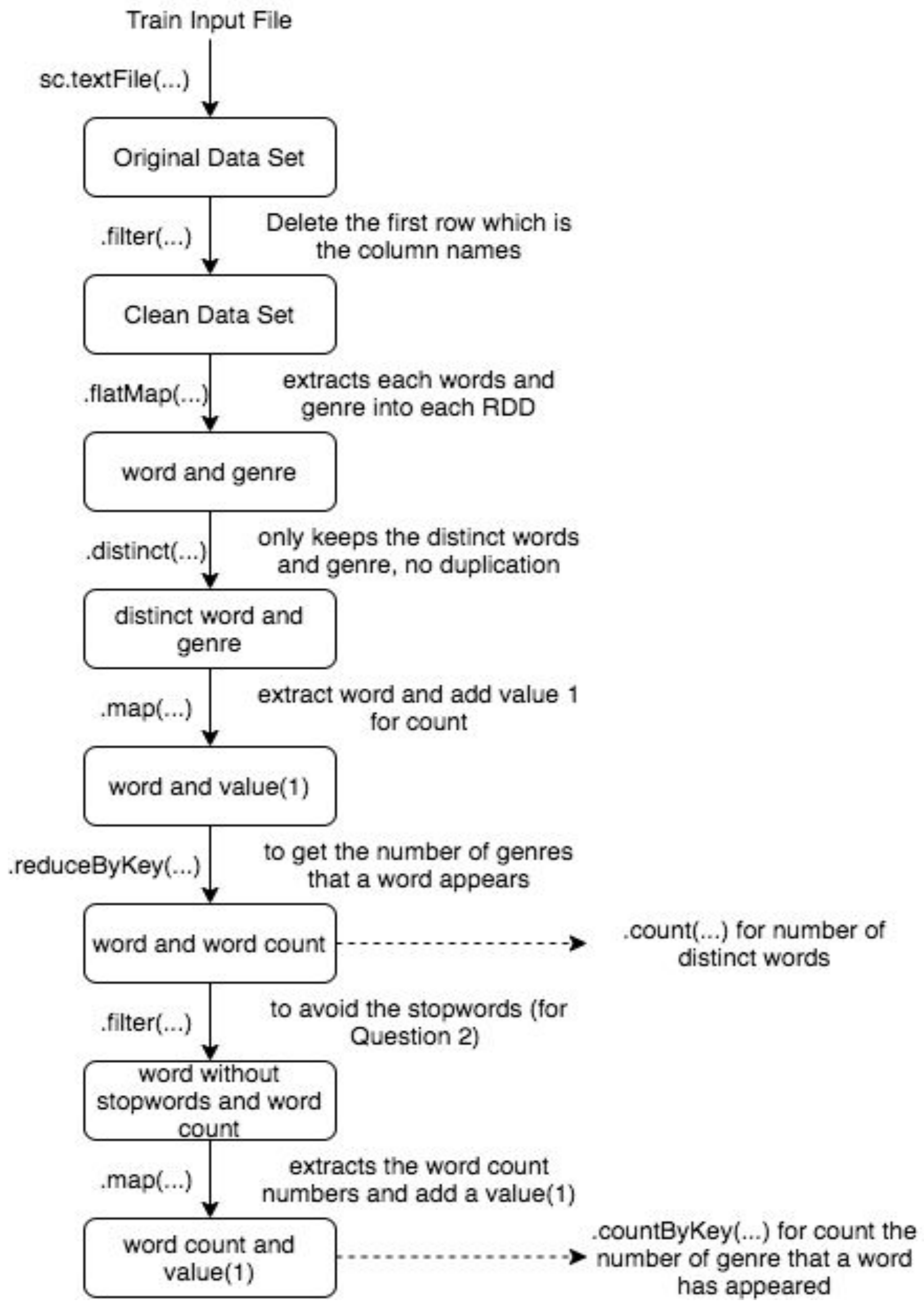
This assignment is implemented by Python and PySpark. The vocabulary exploration part is based on Spark RDD API and it needs to import nltk to extract words from sentences. For sentence vector exploration, we designed to use SparkSQL and SparkML. Packages like tensorflow, tensorflow hub, kmeans, tf-idf will be used in this part to support our analysis. We debug through EMR Jupyter Notebook and the whole assignment runs on the AWS EMR cluster.

Vocabulary Exploration:

Data flow DAG for part A of vocabulary exploration:


Test Mismatched Input File
sc.textFile(...)
Original Data Set
.filter(...)
Delete the first row which is the column names
Clean Data Set
.map(...)
extracts sentences from each record
sentences
.flatMap(...)
extracts words from the sentences and make them into lower case
word
Combine the words from two data set
Development Mismatched Input File
sc.textFile(...)
Original Data Set
.filter(...)
Delete the first row which is the column names
Clean Data Set
.map(...)
extracts sentences from each record
sentences
.flatMap(...)
extracts words from the sentences and make them into lower case
word
Union
word
.distinct(...)
only keeps the distinct words
distinct word
extracts the words in both data set
Test Matched Input File
sc.textFile(...)
Original Data Set
.filter(...)
Delete the first row which is the column names
Clean Data Set
.map(...)
extracts sentences from each record
sentences
.flatMap(...)
extracts words from the sentences and make them into lower case
word
Combine the words from two data set
Development Matched Input File
sc.textFile(...)
Original Data Set
.filter(...)
Delete the first row which is the column names
Clean Data Set
.map(...)
extracts sentences from each record
sentences
.flatMap(...)
extracts words from the sentences and make them into lower case
word
Union
word
.distinct(...)
only keeps the distinct words
distinct word
intersaction
common word

Data flow DAG for part B of vocabulary exploration:


Train Input File
sc.textFile(...)
Original Data Set
.filter(...)
Delete the first row which is the column names
Clean Data Set
.flatMap(...)
extracts each words and genre into each RDD
word and genre
.distinct(...)
only keeps the distinct words and genre, no duplication
distinct word and genre
.map(...)
extract word and add value 1 for count
word and value(1)
.reduceByKey(...)
to get the number of genres that a word appears
word and word count
.count(...) for number of distinct words
.filter(...)
to avoid the stopwords (for Question 2)
word without stopwords and word count
.map(...)
extracts the word count numbers and add a value(1)
word count and value(1)
.countByKey(...) for count the number of genre that a word has appeared

For part A, we read and convert the four files to RDD. For these four parts we first do the same operations. We first delete the first line representing the column name, and then obtain the sentence in each record and further convert it into a word. Then we merge the words in the same type of file, and then get the two types of words to be compared (matched and mismatched). We continue to derive the number of non-repetitive words in each type of data, and get the number of words they have in common. Finally, the final result is obtained by calculation (as shown in the figure below).

For part B, we also delete the first line after reading the file. Then we get word and genre through flatmap, and further get distinct word and genre. We can calculate the number of genre that each word appears in, and we can get the total number of distinct words. Through countByKey, we can get the number of words that appear different times, and finally calculate the ratio required by the answer.

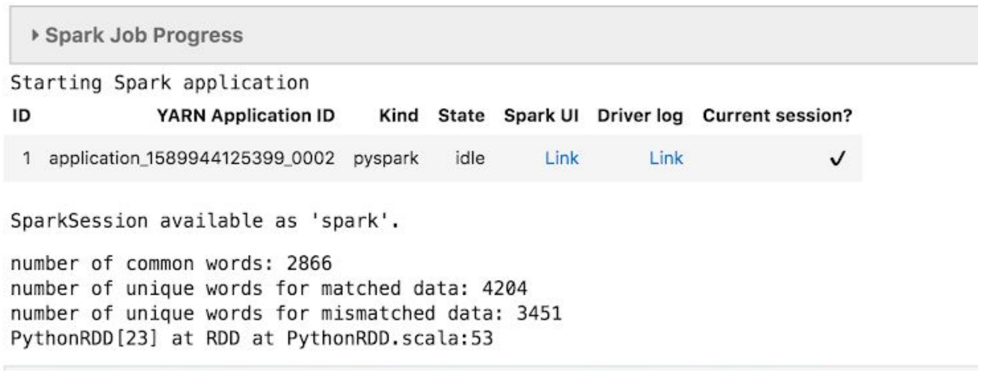
For calculating the result for the part A questions, we get the number of words in both matched and mismatched data set and we have the number of common words. Therefore, to calculate the number of words that just appear in each data set, we do not need additional calculation, but just subtract the number of common words from the number of words in each data set, and we could get the final result.

Because of the lazy evaluation of the Spark, all the transformation will not be executed until some actions need the transformed data. In this application, when we calculate the final result, we need to use .count() which is an action. Hence, it will transform the RDDs for the first time. However, we have three .count() to get the answers for three questions, and each time the application will transform the same bunch of RDDs, which is a resource wasted. So we store the RDDs distinct_words_mat and distinct_words_mismat which are used to calculate the results into the distributed cache by using .persist(), and this will let our application runs nearly three times faster. Also for part b of the vocabulary exploration section, we use .persist() for the RDDs that need to be used several times.

Spark has another important concept named closure. When the application gets lambda functions, it will store the variables of the function and send them to every executor. In this application, we need to use the stopwords information to filter the words. If we simply use .filter() and pass the functions containing the stopwords information, we need to send the stopwords to every task the application has created. To solve this, we use the broadcast variable (sc.broadcast(...)).

The operation distinct (distinct_words_mat = words_mat.distinct(), distinct_words_mismat = words_mismat.distinct()) will be a resource-consuming operation because it incurs shuffling. The distinct operation needs to put all the same words into the same node and sort the words. Similarly, the intersection operation (common_words = distinct_words_mat.intersection(distinct_words_mismat)) will incurs shuffling which would be costly as well. Two RDDs must be operated by the same partitioner for the intersection.

Final Result for part A of vocabulary exploration:



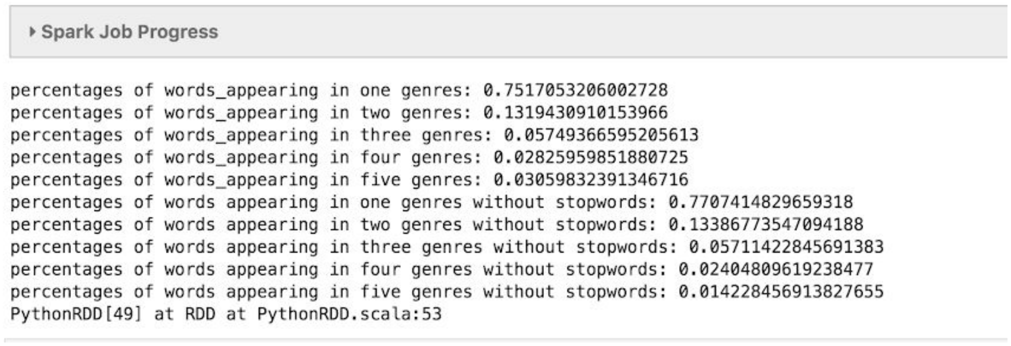
```
Starting Spark application
```

ID	YARN Application ID	Kind	State	Spark UI	Driver log	Current session?
1	application_1589944125399_0002	pyspark	idle	Link	Link	✓

```
SparkSession available as 'spark'.

number of common words: 2866
number of unique words for matched data: 4204
number of unique words for mismatched data: 3451
PythonRDD[23] at RDD at PythonRDD.scala:53
```

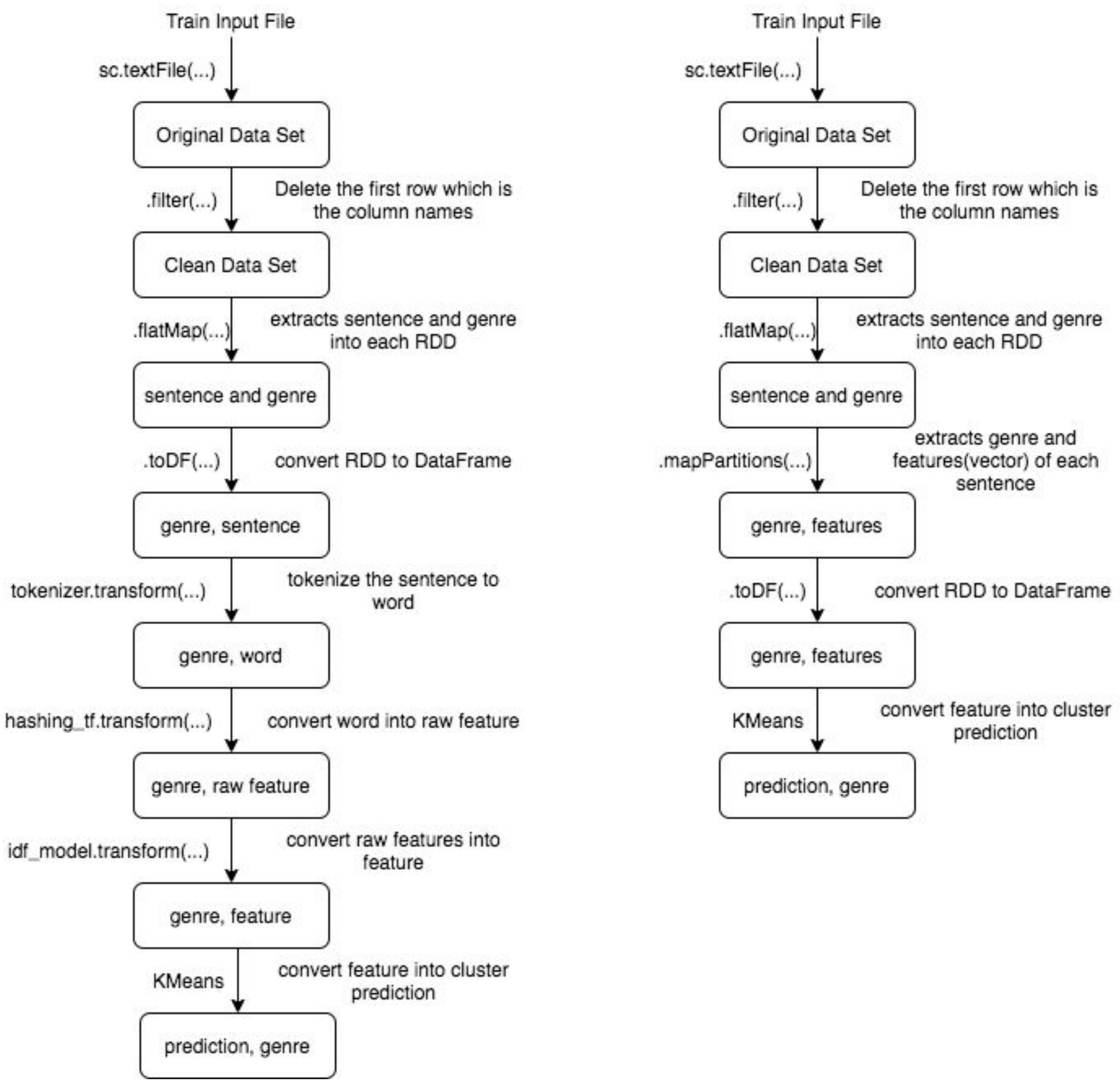
Final Result for part B of vocabulary exploration:



```
percentages of words_appearing in one genres: 0.7517053206002728
percentages of words_appearing in two genres: 0.1319430910153966
percentages of words_appearing in three genres: 0.05749366595205613
percentages of words_appearing in four genres: 0.02825959851880725
percentages of words_appearing in five genres: 0.03059832391346716
percentages of words appearing in one genres without stopwords: 0.7707414829659318
percentages of words appearing in two genres without stopwords: 0.13386773547094188
percentages of words appearing in three genres without stopwords: 0.05711422845691383
percentages of words appearing in four genres without stopwords: 0.02404809619238477
percentages of words appearing in five genres without stopwords: 0.014228456913827655
PythonRDD[49] at RDD at PythonRDD.scala:53
```

Sentence Vector Exploration

Data flow DAG for sentence vector exploration:

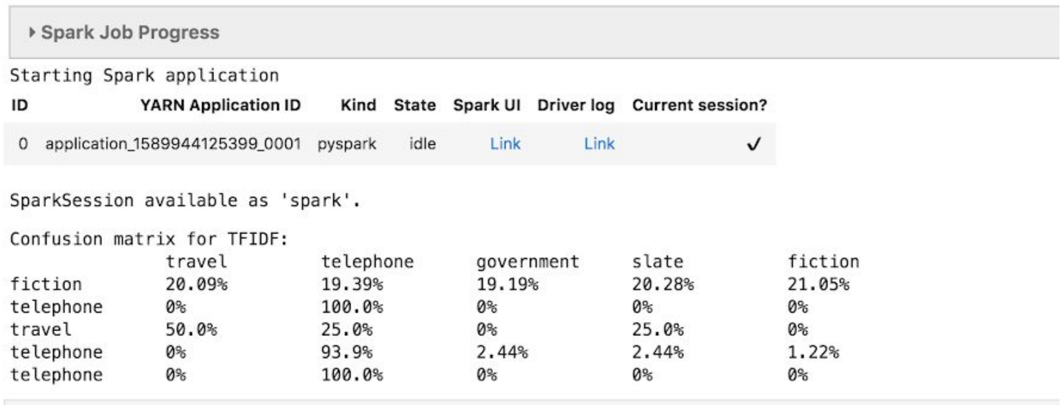


There are 5 steps for using tf-idf based vector representation. First, pair genres with sentences after deleting the first row of the original file. After using textFile to convert the tsv file to rdd, we use a filter to remove the first row of the file and use flatMap method to pair genres and sentences. We cache the result rdd into memory since both 2 parts will use this rdd. Second, we transform it to a dataframe to extract words through tokenizer. Third, hashingtf and idf are applied to transfer vectors. The numFeature is set to 512 since the pretrained encoder has default 512 dimensions. The same number will be better to compare their performance. Only genre and feature columns are kept for further process by using select. Fourth, kmean is used for clustering and only prediction and genre columns are selected. Last, we count the number of words in each genre for every cluster by groupBy and count methods and collect the result since it's very small. The final confusion matrix is calculated through python.

For the pre-trained encoder part, the only difference is the step 2 and step 3. It uses the mapPartition with emr method to convert each partition of the sentence into word vectors. Then transform it into a dataframe. By using udf and withColumn methods we change the data type of features into lists and fit it into kmeans model.

Compare the final result between tf-idf and pre-trained encoder, the performance of tf-idf is better than pre-trained encoder in terms of accuracy. Since tf-idf has 74% of total accuracy where pre-trained only has 50%. (accuracy = Sum each correct clustered genre’s percentage/500) But each cluster in a pre-trained model has one percentage that is much higher than the other where tf-idf has some genres in the same cluster with similar percentages. So the pre-trained model seems more suitable for kmeans clustering algorithm as it puts the same genres in one cluster and each genre has their own cluster. The features that kmeans extract from pre-trained might be more accord with tha specific genre.

Final Result for TFIDF confusion matrix:

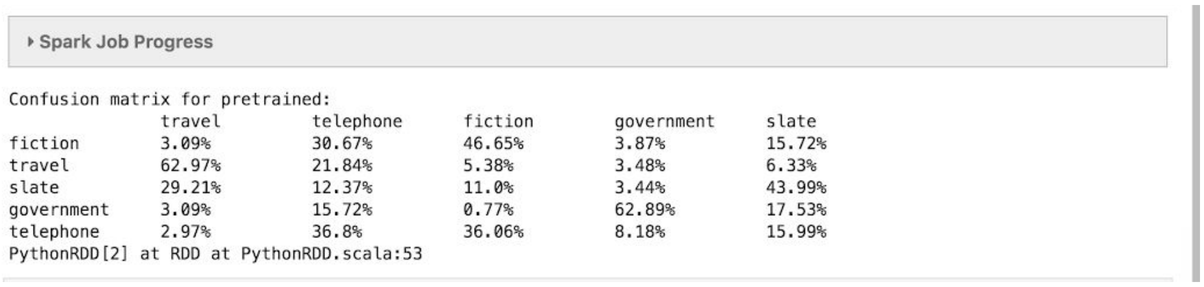


```
▸ Spark Job Progress
Starting Spark application
ID   YARN Application ID               Kind      State   Spark UI   Driver log   Current session?
0    application_1589944125399_0001    pyspark   idle    Link       Link         ✓

SparkSession available as 'spark'.

Confusion matrix for TFIDF:
               travel         telephone      government     slate          fiction
fiction        20.09%         19.39%         19.19%         20.28%         21.05%
telephone      0%             100.0%         0%             0%             0%
travel         50.0%          25.0%          0%             25.0%          0%
telephone      0%             93.9%          2.44%          2.44%          1.22%
telephone      0%             100.0%         0%             0%             0%
```

Final Result for pre-trained sentence encoder confusion matrix:



```
▸ Spark Job Progress

Confusion matrix for pretrained:
               travel         telephone      fiction        government     slate
fiction        3.09%          30.67%         46.65%         3.87%          15.72%
travel         62.97%         21.84%         5.38%          3.48%          6.33%
slate          29.21%         12.37%         11.0%          3.44%          43.99%
government     3.09%          15.72%         0.77%          62.89%         17.53%
telephone      2.97%          36.8%          36.06%         8.18%          15.99%
PythonRDD[2] at RDD at PythonRDD.scala:53
```

Performance Evaluation

We set 2 different running environments for this assignment. One is 5 core nodes with m4.large instance which only has 2 vCPUs and 8GB memory and the other one is 2 core nodes with m4.xlarge instance which has 4 vCPUs and 16GB memory. By setting true for maximizeResourceAllocation, we allow EMR automatically configure spark-default properties. Table 3 shows some execution statistics.

Execution in an environment with high performance is faster than low performance with multiple nodes. Vocabulary exploration part1 runs in 29 seconds with the lower one but runs in 16 seconds with the higher one. The m4.xlarge environment is nearly 50% faster than the m4.large environment. But they have similar performance for part 2 which takes 50 seconds for m4.large and 65 seconds for m4.xlarge. For Sentence vector exploration, both tf-idf model and pre-trained model takes longer than vocabulary exploration. m4.xlarge runs 40 % faster than m4.large. Hence, the environment with higher performance executes programs much faster than the one with lower performance no matter how many nodes there are.

Moreover, we observed the job involves shuffling usually takes longer than the others from the table. Part 2 of vocabulary exploration runs longer than part 1 since the input data is larger as well as the shuffle size. However, job 1 and job 2 of vocabulary part 1 is different. Job 1 takes 6 seconds with 576 KB shuffle size where job 2 only takes 1 second with 888KB shuffle size. The reason why job2 takes less time might be that we persist previous data into memory so job 2 skipped 2 stages and it can use cached data which was calculated before. Sentence pre-trained model also runs longer than sentence tf-idf model with smaller shuffle size.

Question	Job	Stage	Elapsed Time (m4.large)	Executors (m4.large)	Elapsed Time (m4.xlarge)	Executors (m4.xlarge)	Shuffle Read	Shuffle Write
Vocab - part1	0	0	14.18	4	10.5	2		631.1KB
		1	7.26	5	0.75	2	631.1KB	
			Total time : 22		Total time : 11		Total Shuffle size: 631.1KB	
	1	2	5.68	4	3	2		576.1KB
		3	0.62	4	0.4	2	576.1KB	
			Total time : 6		Total time : 3		Total Shuffle size: 576.1KB	
	2	6	0.82	5	0.8	2		887.9KB
		7	0.38	5	0.3	2	887.9KB	
			Total time : 1		Total time : 1		Total Shuffle size: 887.9KB	
Vocab - part2	3	8	53.66	5	42.8	2		15.2MB
		9	1.87	5	0.8	2	15.2MB	2.2MB

		10	0.61	5	0.4	2	2.2MB	
			Total time : 56		Total time : 44		Total Shuffle size: 17.4MB	
	4-14	13-43	average: 0.29	5	average: 0.2	2		
	0	0	17.87	1	14	1		
	1	1	31.51	2	16	1	18.7KB	
		2	0.63	2	0.6	1		18.7KB
			Total time: 32		Total time: 16		Total Shuffle size: 18.7KB	
	2	3	13.76	2	11	1		
Sentence - tfidf	3-28	4-50	average: 0.56	2	average: 0.3	1		
	29	51	11.89	2	9	1	2.1KB	
		52	0.82	2	0.6	1		2.1KB
			Total time: 13		Total time: 9		Total Shuffle size: 2.1KB	
	30	53	11.19	2	8	1	11KB	
		54	0.77	2	0.6	1		11KB
			Total time: 12		Total time: 9		Total Shuffle size: 11KB	
	0	0	36	1	30	1		
	1	1	36	1	26	1		
	2	2	22	1	22	1		
Sentence - pretrain	3-24	3-49	0.15	1	0.1	1		
	25	50	47	2	25	1		622B
		51	1	1	0.5	1	622B	
	26	52	84	2	27	1		3.6KB
		53	2	1	0.5	1	3.6KB	

table 3

Conclusion

Throughout the assignment implementation, we found that there is a close relationship between shuffle size and execution time. As the shuffle size increase, the execution time will rise. Hence, cache the repeated data into memory plays a significant role in reducing execution time. Furthermore, the universal sentence encoder is more suitable for kmean clustering based on our result but the tradeoff is the running time which could be doubled of the tf-idf model.