Indian Institute of Technology Madras

BIG DATA LAB FINAL PROJECT

Group 1: DataBenders.V2

M V A Suhas Kumar EE17B109
Om Shri Prasath EE17B113
Pruthvi Raj R G EE17B114

Objective

In this project, we are required to build a real-time Kafka Spark integration that takes in the data from a given link and produces the predictions on this data. This project serves as a final exercise in applying the concepts we have learned throughout the course. We will be using Dataproc for Apache Spark workspace and Kafka VMs available on GCP. Our aim is divided into four major parts :

a. Data Exploration
b. Data Preprocessing
c. Training
d. Real-time Inference

We will be using the Jupyter notebook web interface available in Dataproc clusters for ease of use.

Data Exploration

We are given the data of hotel reviews from the “YELP” dataset. We will be using the “text”(which contains the review in text format) and “stars” columns in the given dataset as we believe the rest of the columns do not contribute heavily to the accuracy of the model. We will be exploring the “text” column under various settings of “stars” to understand the data better.

1. **Basic data attributes**

Raw data:

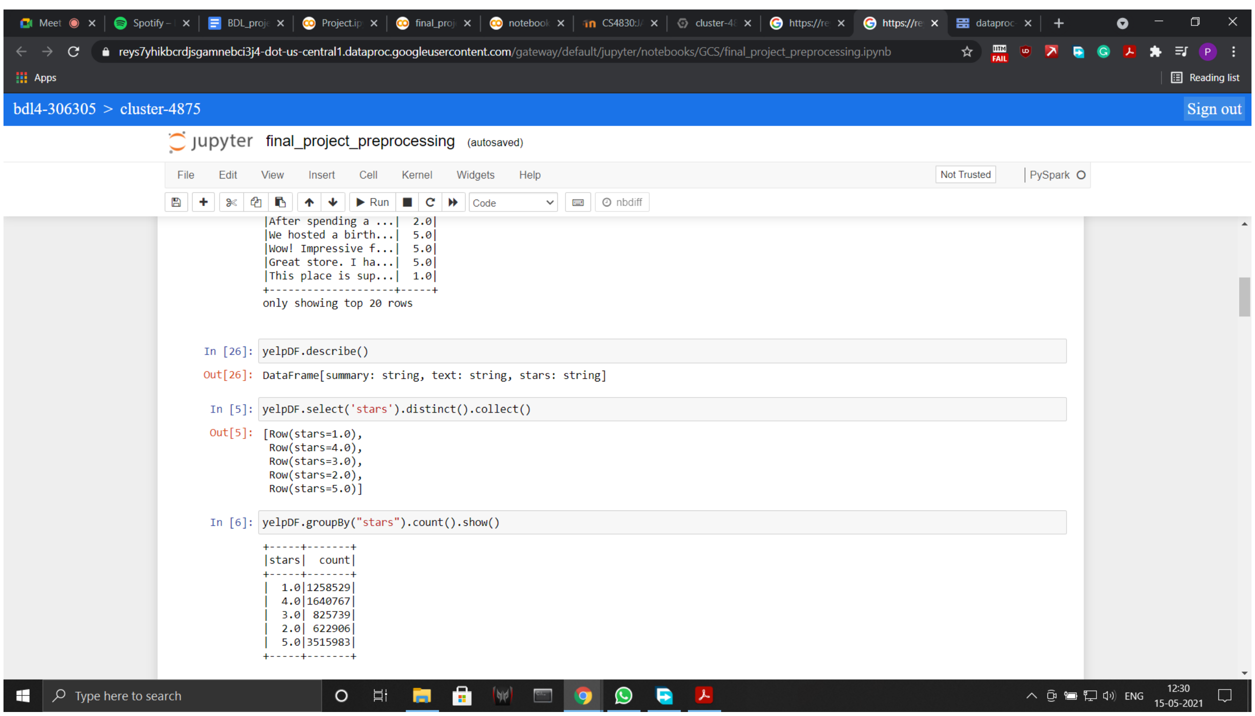
```
+--------------------+-----+
|                text|stars|
+--------------------+-----+
|I had my sofa, lo...|  5.0|
|Again great servi...|  5.0|
|Opening night, ne...|  4.0|
|Fun times. Great ...|  4.0|
|I wanted to like ...|  2.0|
|Someone with my c...|  2.0|
|If you're looking...|  1.0|
|My friend won a f...|  5.0|
|My least favorite...|  2.0|
|This place is sup...|  1.0|
+--------------------+-----+
```

We see that we have 5 types of reviews, ranging from 1 to 5.

```
[Row(stars=1.0),
 Row(stars=4.0),
 Row(stars=3.0),
 Row(stars=2.0),
 Row(stars=5.0)]
```

We observe that there is a class imbalance in the reviews.

```
+-----+-------+
|stars|  count|
+-----+-------+
|  1.0|1258529|
|  4.0|1640767|
|  3.0| 825739|
|  2.0| 622906|
|  5.0|3515983|
+-----+-------+
```



2. **The average length of review based on stars**

The mean length of the reviews

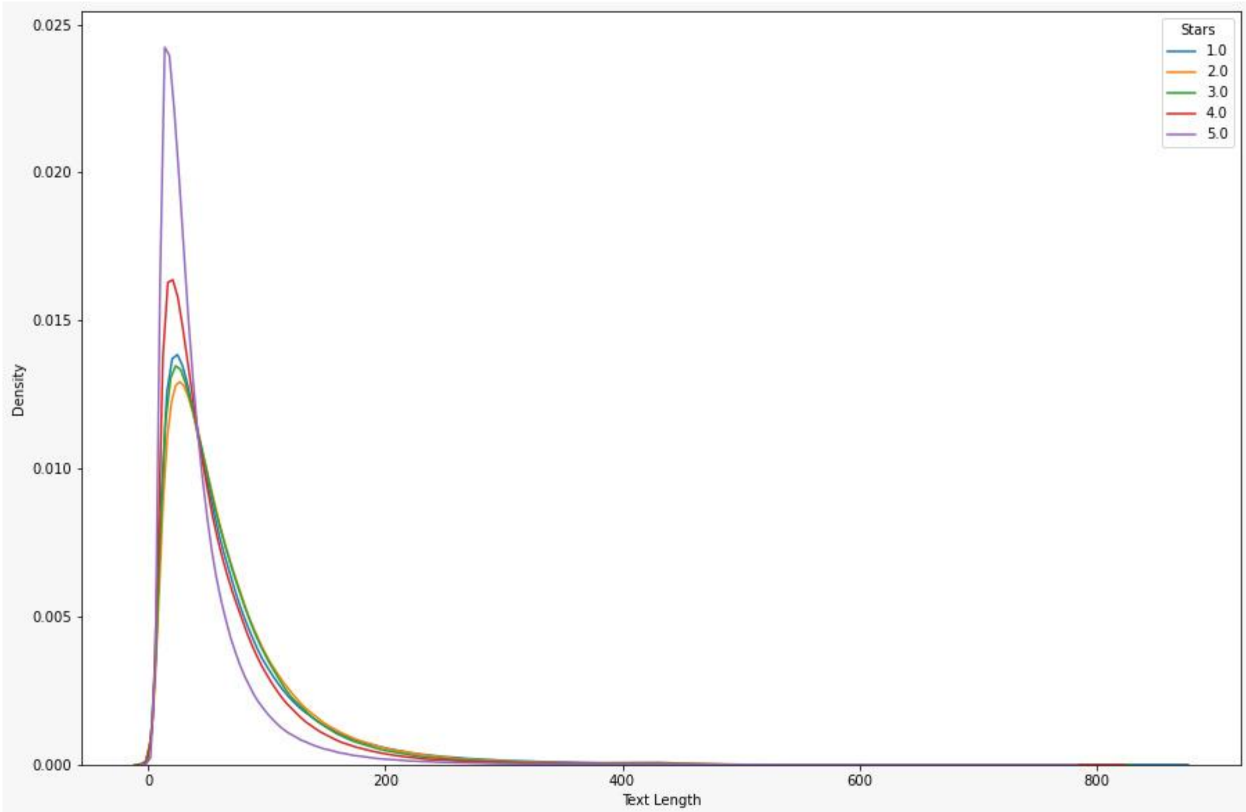
stars	avg(len)
1.0	72.8403509176189
4.0	62.96502733172961
3.0	71.03799142344009
2.0	74.71956121790447
5.0	47.56868960970517

The standard deviation of the length of reviews

stars	stddev_samp(len)
1.0	67.23738578012967
4.0	55.93379165381655
3.0	60.544811349643254
2.0	64.15716298670213
5.0	45.22442870105579

We observe that the longer the text poorer is the review. This makes practical sense as we try to rant about the hotel in the reviews if we do not like the service. But we rarely write long reviews if we like it. However, the variance in the sentence length is extremely high, hence, we think that this feature will not be very helpful in building the model. (We have removed the special characters before finding the length of reviews)

The following graph shows the density(the graphs are normalized) of various classes, with the length of review on the x-axis.

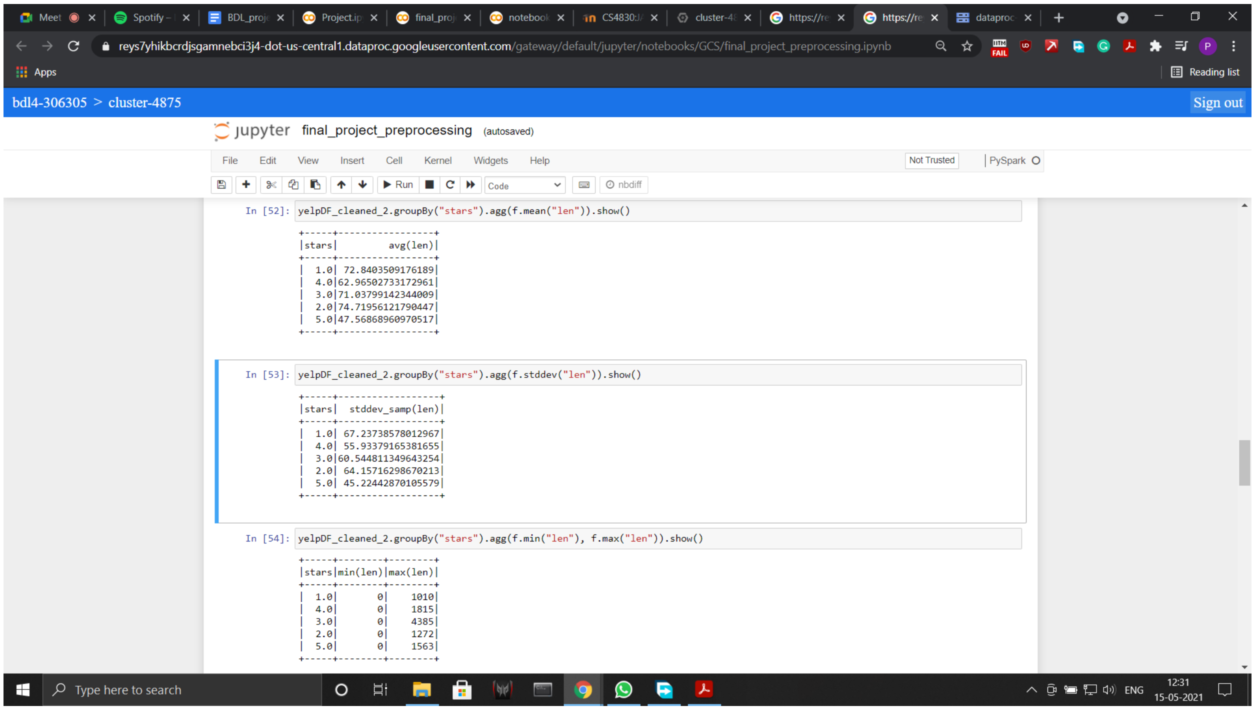


3. **The minimum and maximum length of reviews based on stars**

   After removing the special characters, we found out the minimum and maximum lengths of the reviews to understand the limits.

stars	min(len)	max(len)
1.0	0	1010
4.0	0	1815
3.0	0	4385
2.0	0	1272
5.0	0	1563

We observe that they are empty reviews ie, the person gave the star without writing a descriptive review. This will be challenging for the model to handle as there is no text data to base the prediction on!

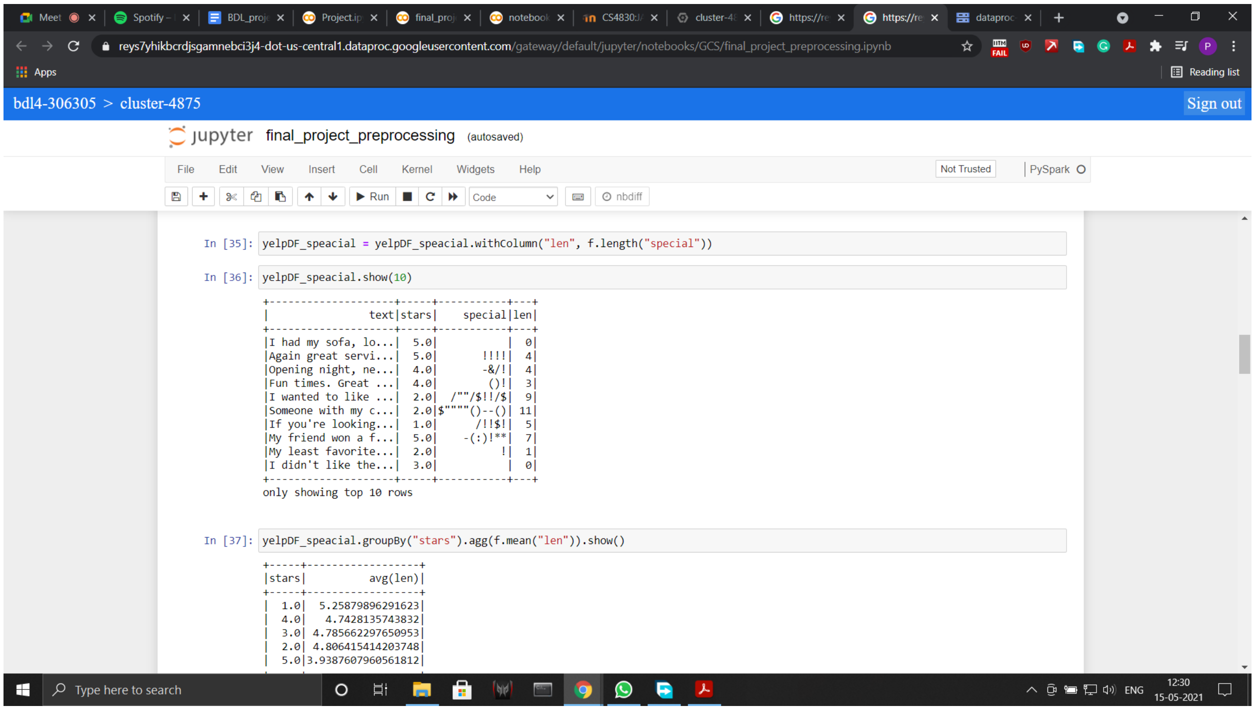


4. The average length of special characters based on stars

Most of the reviews consist of special characters like “!”, “?” and “*”. We hypothesized that the number of special characters used will be higher in two cases: when the person liked the service very much or he hated it. Here are our findings.

```
+-----+------------------+
|stars|          avg(len)|
+-----+------------------+
|  1.0|  5.25879896291623|
|  4.0|   4.7428135743832|
|  3.0| 4.785662297650953|
|  2.0| 4.806415414203748|
|  5.0|3.9387607960561812|
+-----+------------------+
```

We observe that this is partially true, people used a lot more special characters when they were extremely unhappy with the service. On the contrary, they used very less special characters when they loved the hotel(5 stars)



5. **Most occurring unigram and bigrams based on the type of review**

Next, we wanted to explore the most occurring words in the reviews, based on the stars. Instead of doing this separately for each star class, we decided to combine 4 and 5 stars in a single class called the "positive" class and the rest 1, 2, 3 stars into the "negative" class. Initially, we tried to find the top "unigrams" in both the classes and found that they were overlapping a lot. Hence, we tried to find the top "bigrams" instead. By "top" bigrams we mean the most occurring. Here are the findings:

Unigrams for the positive class

```
+----------+---+-------+
|     words|pos|  count|
+----------+---+-------+
|     place|  1|2400894|
|      good|  1|2232807|
|     great|  1|2197960|
|      food|  1|2124657|
|      time|  1|1556942|
|   service|  1|1436538|
|      like|  1|1397743|
|       get|  1|1313742|
|      back|  1|1287812|
|       one|  1|1192169|
|    really|  1|1132196|
|        go|  1|1099093|
|      also|  1| 968418|
```

always	1	909719
best	1	904205
nice	1	879759
friendly	1	876579
well	1	813898
staff	1	795331
us	1	794262
delicious	1	788626
amazing	1	788065
got	1	782416
love	1	776702
definitely	1	738532
try	1	697798
recommend	1	693124
little	1	685126
even	1	643336
restaurant	1	641189
come	1	626379

Unigrams for the negative class

words	pos	count
food	-1	1450760
place	-1	1279943
good	-1	1217210
like	-1	1199459
get	-1	1199301
time	-1	1132565
back	-1	1057516
one	-1	1045730
service	-1	1045225
us	-1	824406
go	-1	821691
even	-1	705435
got	-1	697469
said	-1	694942
order	-1	646182
really	-1	645767
told	-1	609534
came	-1	576162
ordered	-1	575204
never	-1	554784
asked	-1	514411
people	-1	509772
great	-1	495156
minutes	-1	494160

went	-1	488309
come	-1	451243
know	-1	447584
much	-1	445384
going	-1	439954
restaurant	-1	437747
2	-1	434264

Bigrams for the positive class

words	sum(pos_count)
highly recommend	194890
really good	116380
definitely back	94015
Las Vegas	91616
great service	85334
first time	83969
one best	81836
ice cream	80065
staff friendly	77195
food great	77191
love place	75198
great place	74851
service great	74068
Highly recommend	70988
come back	70942
definitely recommend	67574
great food	67257
5 stars	66628
next time	66076
great experience	61424
super friendly	59551
recommend place	57524
great job	57487
make sure	56650
Great food	56231
every time	55824
friendly staff	55481
Great service	54923
Great place	53788
coming back	53155
happy hour	50944
definitely come	50258
friendly helpful	47442
good food	46929
food delicious	46810
top notch	45484
place great	45051
Love place	44768
back try	43689

really enjoyed	42665
go wrong	41578
place go	41051
food amazing	39879
wait go	39843
definitely go	38186
well worth	37922
best ever	37637
made sure	37330
really nice	36956
great time	36745

Bigrams for the negative class

words	sum(pos_count)
tasted like	-39397
20 minutes	-39066
10 minutes	-38397
15 minutes	-34945
minutes later	-32226
somewhere else	-31669
call back	-30697
waste time	-30105
told us	-29498
came back	-29418
3 stars	-28774
30 minutes	-28575
customer service	-28330
2 stars	-27162
looked like	-25559
nothing special	-25164
credit card	-24546
never go	-24116
front desk	-23856
1 star	-20707
one star	-19880
45 minutes	-19119
minutes get	-18619
last time	-18605
someone else	-17644
money back	-16769
never came	-16683
5 minutes	-16548
called back	-16488
walked away	-16447
two stars	-16249
waste money	-16245
take order	-15698
time money	-14681
give another	-14677

```
|    Needless say|        -14323|
|    never return|        -14288|
|    poor service|        -14193|
|        get food|        -14179|
|    took forever|        -13783|
|      zero stars|        -13598|
|         food ok|        -13570|
|        got home|        -13496|
|      never come|        -13361|
|    service ever|        -13316|
|     bad service|        -13171|
|     finally got|        -12946|
|service terrible|        -12748|
|   speak manager|        -12575|
|  extremely rude|        -12540|
+----------------+--------------+
```

(The (-) sign before the counts are just for showing that they are from the negative class)
We clearly see that the bigrams bring out the descriptive wordings of both classes very well.

Preprocessing

Now that we have explored the data, we need to prepare the data to feed into our model. As this is a text-based model, we need to convert the text into a numeric format to use it in our model. We have listed the preprocessing steps we followed below:

1. **Special character removal:**
   We have seen that many reviews have some special characters in them. These special characters can be used to make emojis[:) :(;) etc] which cannot be directly interpreted when we convert them into a numeric format. Hence, it is essential to remove these special characters from each review.

```
+--------------------+-----+-----------+---+
|                text|stars|    special|len|
+--------------------+-----+-----------+---+
|I had my sofa, lo...|  5.0|           |  0|
|Again great servi...|  5.0|       !!!!|  4|
|Opening night, ne...|  4.0|       -&/!|  4|
|Fun times. Great ...|  4.0|        ()!|  3|
|I wanted to like ...|  2.0|  /""/$!!/$|  9|
|Someone with my c...|  2.0|$""""()--()| 11|
|If you're looking...|  1.0|      /!!$!|  5|
|My friend won a f...|  5.0|    -(:)!**|  7|
|My least favorite...|  2.0|          !|  1|
|I didn't like the...|  3.0|           |  0|
+--------------------+-----+-----------+---+
```

2. **Stopwords removal**
   After removing the special characters, we are left with only English words. We understand that not all words in the sentence are essential to understand the essence of the sentence. For example, the words “and”, “is”, “I”, “then” do not give any information about the class it belongs to. Hence, it is essential that we remove these ‘stopwords’ that interfere with the model building.

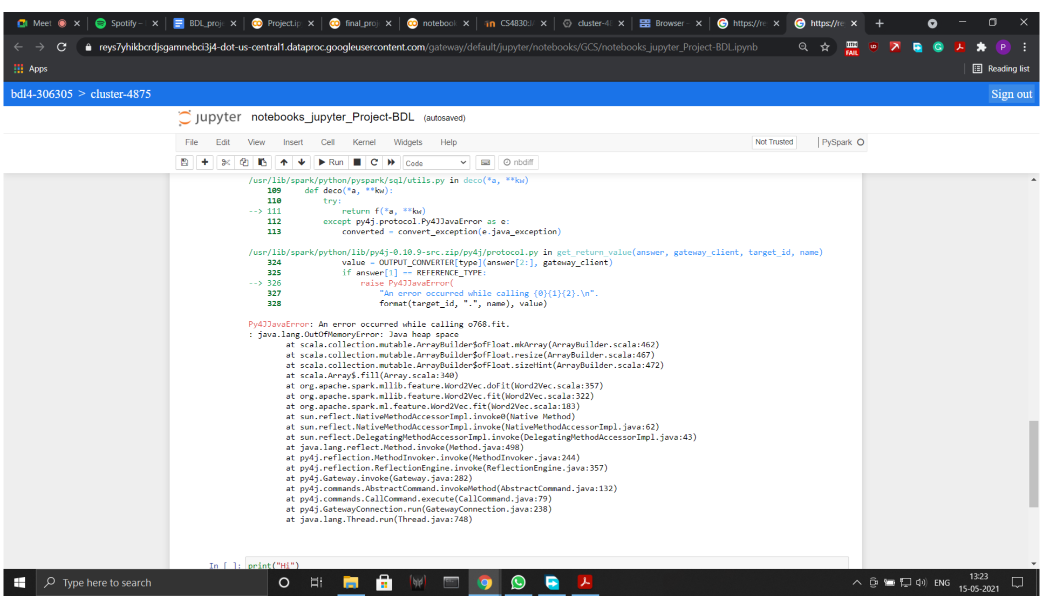
3. **Word tokenization**
   This step just splits the words in the sentence into an array of words. Later we will be able to use this array of words for further processing. After the words are cleaned and tokenized, the Spark dataframe looks like it is shown below:

```
+--------------------+-----+--------------------+--------------------+
|                text|stars|               words|         clean_words|
+--------------------+-----+--------------------+--------------------+
|I had my sofa lov...|  5.0|[i, had, my, sofa...|[sofa, love, seat...|
|Again great servi...|  5.0|[again, great, se...|[great, service, ...|
|Opening night new...|  4.0|[opening, night, ...|[opening, night, ...|
|Fun times Great a...|  4.0|[fun, times, grea...|[fun, times, grea...|
|I wanted to like ...|  2.0|[i, wanted, to, l...|[wanted, like, pl...|
|Someone with my c...|  2.0|[someone, with, m...|[someone, company...|
|If youre looking ...|  1.0|[if, youre, looki...|[youre, looking, ...|
|My friend won a f...|  5.0|[my, friend, won,...|[friend, won, fre...|
|My least favorite...|  2.0|[my, least, favor...|[least, favorite,...|
|I didnt like the ...|  3.0|[i, didnt, like, ...|[didnt, like, ban...|
+--------------------+-----+--------------------+--------------------+
```

Here the “words” column is an array of words after special character removal, and the column “cleaned_words” is the array of words after removing the stop words too.

4. Word2Vec

As we mentioned earlier, we need a method to convert the “textual” features into “numeric” features. Word2Vec is a method that allows us to do this. We will encode the words into “embeddings”. The word2vec model requires a hyperparameter to be set, the size of the word embedding to train. We observed that as we increased the size of the embeddings from 50 to 1000, the model accuracy increased drastically, from ~54% to ~65%. However, we could not increase the size of the embedding vectors further as we got an out-of-memory error.



5. **TF-IDF**

   This is another method that we can use to convert the textual features into numeric ones. This method relies on the "frequentist" approach, that words occur with different frequencies between classes. We need to choose the type of TF-IDF - "unigrams", "bigrams" etc. We explored both cases. **We got a higher accuracy with unigrams(~67%), against bigrams(~54%). We also played with the number of words in the dictionary attribute of TFIDF and we found that keeping a very high number(10,000) helps us get better accuracy.** The dataframe looks like as shown below after applying the transform.

```
+--------------------+-----+
|            features|stars|
+--------------------+-----+
|(10000,[698,712,3...|  5.0|
|(10000,[142,750,1...|  4.0|
|(10000,[6,41,1376...|  1.0|
|(10000,[86,115,28...|  4.0|
|(10000,[444,495,5...|  5.0|
|(10000,[183,204,4...|  3.0|
|(10000,[1086,1395...|  4.0|
|(10000,[44,129,28...|  5.0|
|(10000,[29,281,36...|  3.0|
|(10000,[157,266,4...|  5.0|
+--------------------+-----+
```

Training

After we have converted the review into numeric features, it is time to use them to build an ML model. To do this, we create a pipeline that will apply the transformations that we have discussed earlier on the data. We also split the data into train and test sets to get an estimate of the model performance on unseen data. We will also be able to use the model we like, for example, logistic regression, random forest, decision trees, etc. We can later evaluate the model using a score, for example, accuracy, F1 score, or print the confusion matrix. After building the model we will be able to predict any new data as shown below:

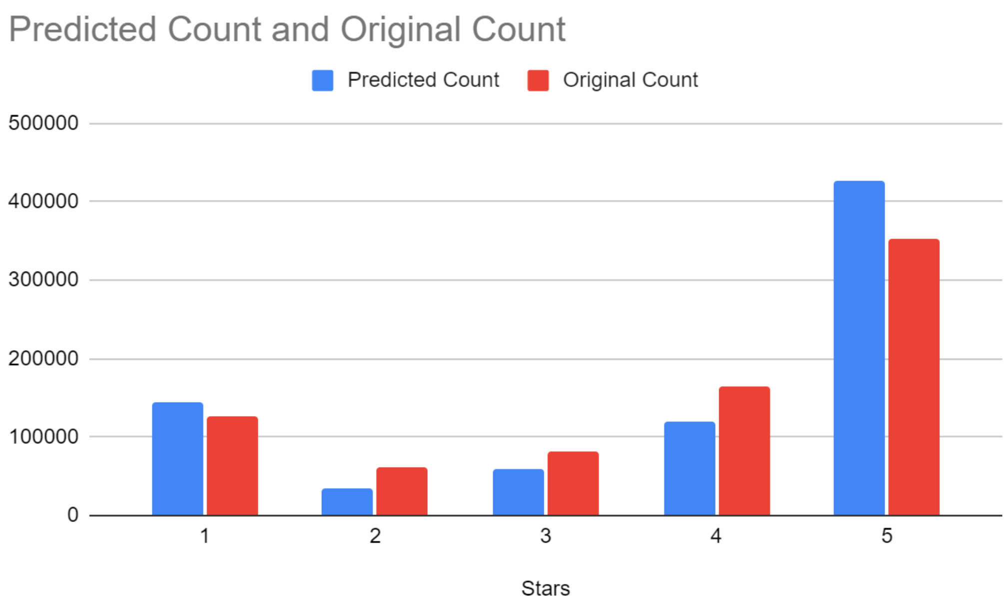
```
+----------+-----+
|prediction|stars|
+----------+-----+
|       5.0|  5.0|
|       5.0|  4.0|
|       1.0|  1.0|
|       3.0|  4.0|
|       5.0|  5.0|
+----------+-----+
```

Original Data's distribution

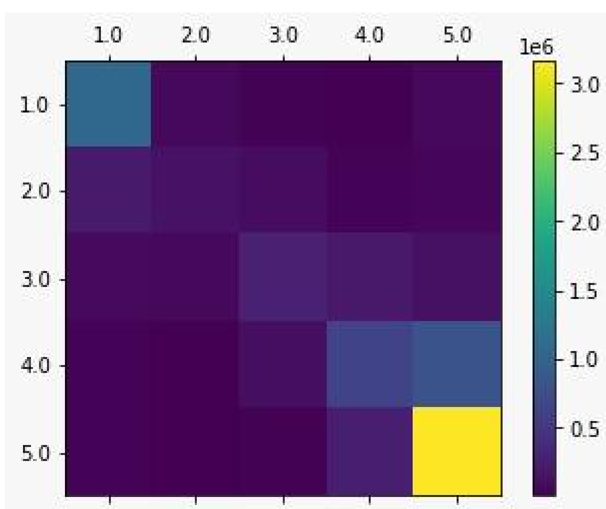
```
+-----+------+
|stars| count|
+-----+------+
|  1.0|125919|
|  4.0|163794|
|  3.0| 82400|
|  2.0| 62410|
|  5.0|352282|
+-----+------+
```

Predicted Data's distribution

```
+----------+------+
|prediction| count|
+----------+------+
|       1.0|145275|
|       4.0|120250|
|       3.0| 58964|
|       2.0| 35169|
|       5.0|427147|
+----------+------+
```



Inference: We can see that the model is getting confused between (class 4 and class 5), (class 1 and class 2), etc. This shows us that the model is pushing the predictions to their "extremes" , that is if the review is slightly good (around 3 or 4 stars), it is predicting the stars to be 5. On the other hand, if the review is slightly worse, the prediction is pushed down to 1 star. This can be clearly seen by seeing the counts of class 1 and class 5 in the above plot. It can also be seen from the below confusion matrix generated on the whole data between actual and predicted stars.



jupyter training Last Checkpoint: 6 minutes ago (autosaved)

In [16]:
```
yelpDF.printSchema()
```

```
root
 |-- review_id: string (nullable = true)
 |-- text: string (nullable = true)
 |-- stars: double (nullable = true)
 |-- text_raw: string (nullable = true)
 |-- text_split: array (nullable = true)
 |    |-- element: string (containsNull = true)
```

Creating the Pipeline Model

In [18]:
```
remover = StopWordsRemover(inputCol="text_split", outputCol="clean_words") # removing the stop words
hashingTF = HashingTF(inputCol="clean_words", outputCol="rawFeatures", numFeatures=10000) # making features
idf = IDF(inputCol="rawFeatures", outputCol="features", minDocFreq=5)
lr = LogisticRegression(labelCol="stars",featuresCol="features") # logistic regression

(trainingData, testData) = yelpDF.randomSplit([0.8, 0.2]) # splitting train and test
```

Training and prediction

In [19]:
```
pipeline = Pipeline(stages=[remover,hashingTF,idf,lr]) # pipeline with 4 stages as said above

pipeline_model = pipeline.fit(trainingData) #fitting the model
predictions = pipeline_model.transform(testData) #predictions on test data

evaluator = MulticlassClassificationEvaluator(
    labelCol="stars", predictionCol="prediction")
accuracy = evaluator.evaluate(predictions,{evaluator.metricName: "accuracy"}) # accuracy calculation
print("Test Accuracy = %g" % (accuracy))

f1 = evaluator.evaluate(predictions,{evaluator.metricName: "f1"}) # f1 score calculation
print("Test F1 = %g" % (f1))
```

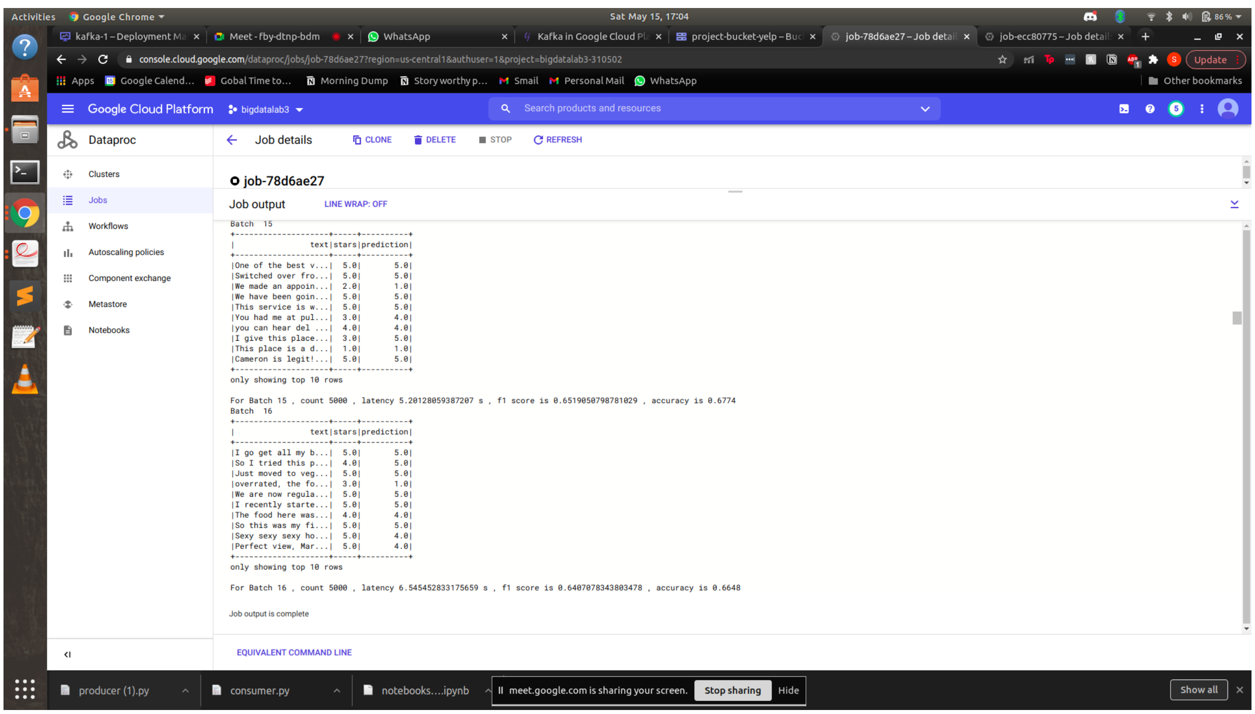
```
Test Accuracy = 0.683716
Test F1 = 0.66193
```

Real-time inference

After building our ML model and saving it, we need to use it to do real-time inference on the data that is being streamed from Kafka. The next task was to build the producer, which will read the data from a given link and publish the data into an IP address which can then be used up by spark to do the prediction. We will then need to write the code to receive this data from Kafka and run inference, this will be submitted as a dataproc job.

The following screenshots show the working of the whole pipeline, the latency of processing, etc. We have printed the accuracy, F1 score, and latency calculated for each of the batches on the console.

We are measuring the latency as follows: "time that is taken for computing the predictions on one batch of data"



So our **real-time inference pipeline** is as follows :

- We use a Spark Producer to write the dataset as messages to a Kafka cluster using Structured Streaming in JSON format
- We read the messages using Spark Consumer using Structured Streaming and parse the JSON string to get the data in the original format
- We extract the text from the data and apply preprocessing to the text as follows
    - We remove the special characters from the text
    - We tokenize the text into words
- We then send the preprocessed words to PipelineModel for prediction which consists as follows :
    - StopWordsRemover
    - HashingTF
    - IDF
    - LogisticRegression
- For the project, we extract the predictions from the data and compute the Accuracy and F1 score for each batch for displaying the model performance.

Conclusion

We were able to do the real-time inferno with a latency of ~6 sec (for a batch size of 5000), and the best model was able to achieve a test accuracy of ~67% and an F1 score of ~65%.
We were able to understand the data well and use the techniques taught in the course to set up the whole pipeline for real-time inference using Kafka and Spark Streaming integration.

File submitted

Producer.py - Code for Kafka Producer
Consumer.py- Code for the consumer used in streaming
Training.ipynb - Training code
Data_exploration.ipynb - Code used for exploration
Unigram.ipynb - Code to generate top unigrams
Bigram.ipynb - Code to generate top bigrams