CS 6350
ASSIGNMENT 2

Please list clearly all the sources/references that you have used in this assignment.

- https://archive.ics.uci.edu/dataset/228/sms+spam+collection
- https://stackoverflow.com/questions/37405617/how-to-use-tf-idf-with-naive-bayes
- https://theflyingmantis.medium.com/text-classification-in-nlp-naive-bayes-a606bf419f8c
- https://www.cs.rhodes.edu/~kirlinp/courses/ai/f18/projects/proj3/naive-bayes-log-probs.pdf

1 Friend Recommendation using Mutual Friends

Link to Databricks notebook:
https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/4636605037478694/3881298198240995/2046848112049253/latest.html

Initial RDD contains: `['0\t1,2,…', '1\t0,5,…', …]` i.e. UserID List of friends' UserIDs separated by commas.

1. Create RDD of pairs of friends i.e. `[(user1, friend1), (user1, friend2), (user1, friend3), …]` by using flat map to map each user to each one of its friends.

2. Join the RDD with itself (self join the RDD). The RDD will contain pairs of friends of friends (they are not friends but share a mutual friend) i.e. `[(user1, (friend1, friend1)), (user1, (friend1, friend2)), …]`.

3. Map the RDD to contain only pairs of friends of friends i.e. take out the user `[(friend1, friend1), (friend1, friend2), …]` and filter out friends with themselves i.e. `(friend1, friend1)`.

4. Subtract the friends RDD from the friends of friends RDD. This will remove pairs who are already friends from the RDD. If the pair is already friends, we do not want to recommend them to each other.

5. Map every pair of friends of friends to a count of 1 and reduce by key. This will give the count of mutual friends the pair has in common i.e. `[(friend1, friend2), count), …]`.

6. Map the RDD to `[(friend1, (friend2, count)), …]`. Each entry now contains a friend recommendation.

7. Group the recommendations by key `[(friend1, [(friend2, count), (friend3, count), (friend4, count), …], (friend2, [(friend1, count), (friend3, count), (friend4, count), …], …]`. Each entry's value has a list of friend recommendations and their counts.

8. Map the RDD to `[(friend1, [top 10 friend recommendations], …]` by sorting each list of friend recommendations and their counts `[(friend2, count), (friend3, count), (friend4, count), …]` by ascending friend ID and descending count, mapping to a list of friend IDs `[friend2, friend3, friend4, …]`, and indexing the first 10 entries in the list (or fewer if there are less than 10 friend recommendations for some users).

2 Implementing Naive Bayes Classifier using Spark MapReduce

Link to Databricks notebook:
https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/4636605037478694/3933656936038108/2046848112049253/latest.html

The dataset used is from https://archive.ics.uci.edu/dataset/228/sms+spam+collection. The SMS Spam Collection is a public set of SMS labeled messages. It is a text based dataset of SMS messages corresponding to a classification problem with two classes 'ham' and 'spam'. In the text file, each line has the correct class followed by the raw message.

Naive Bayes algorithm on the training set: preprocessed training set RDD contains `[('class', 'SMS message'), …]`.

1. Filter the RDD to class ham and get the count. This is the number of SMS messages that have been classified as 'ham'.

2. Filter the RDD to class spam and get the count. This is the number of SMS messages that have been classified as 'spam'.

3. Get the count of total SMS messages in the RDD.

4. Calculate the prior of each class. The prior of class ham P('ham') is the count from (1) divided by the count from (3). The prior of class spam P('spam') is the count from (2) divided by the count from (3).

5. Flat map the RDD to contain `[(('class', 'word'), 1), …]`. This breaks the SMS messages into an individual word with count 1 in each entry in the RDD.

6. Filter the RDD to class ham and get the count. This is the total count of all words in all SMS messages classified as ham.

7. Filter the RDD to class spam and get the count. This is the total count of all words in all SMS messages classified as spam.

8. Reduce the RDD by key. This will give the count of how many times each word has appeared in its class i.e. `[(('class', 'word'), count), …]`.

9. Convert the RDD to a list. This will allow us to use the word counts when we map the test set RDD to a function that classifies the SMS messages in the test set.

10. Get the vocabulary size (number of unique words in the training set) by creating a dictionary with the word as key and None as value, then getting its length.

11. Map the test set RDD to contain `[('class', 'predicted class'), …]` for every SMS message. A function is called in the mapper with the SMS message as argument that returns the predicted class.

How to calculate P(class|SMS) i.e. P('spam'|SMS) and P('ham'|SMS):
1. Split the SMS message into words.
2. For each word in the SMS, filter the training set list from (9) to the class and the word.
3. If the entry is not empty, assign the word count from the training set list. Otherwise, if the entry is empty i.e. [], the count is assigned 0.

To get P(class|SMS), we can multiply P(w|class) for each word w in the SMS then multiply by P(class) where P(w|class) = count of how many times the word w has appeared in that class from (11.3) plus 1 divided by the total count of all words that have been mapped to that class from (6) and (7) plus the vocabulary size from (10). This uses Laplace Smoothing also known as add-1 smoothing, so we can account for words in the test set we have never seen before and words that only appear in one class. For example, "urgent" may appear only in class "spam" in the training set, making P("urgent"|"ham") = 0 and resulting in multiplication by 0 and/or sum of log(0) which is an error in the calculation for P("ham"|SMS). Since probabilities are between 0 and 1, the product starts approaching 0 so log-probabilities are used instead.

4. To get P(class|SMS), we take the sum of each log(P(w|class)) for each word w in the SMS then add log(P(class)).
5. The higher of P('ham'|SMS) and P('spam'|SMS) is the predicted class.

Summary of results:

```
Prior of class ham: 0.871
Prior of class spam: 0.129
Accuracy of model: 0.974

Precision of model: 0.899
Recall of model: 0.921
```

Most of the SMS messages are classified as ham with only a small percentage classified as spam. Accuracy is a measure of the fraction of all SMS messages that are predicted correctly ('ham' predicted as 'ham' and 'spam' predicted as 'spam'). The Naive Bayes model has a high accuracy of 0.974. However, since the classes are unbalanced ('ham' occurs much more frequently than 'spam'), accuracy is not a good metric to look at by itself. In addition to accuracy, precision and recall should be examined to get a better picture of the predicted classification for 'spam'. Precision is true positives/(true positives + false positives) or a measure of when the prediction is spam, how often it is actually spam. Recall is true positives/(true positives + false negatives) or a measure of when the SMS is actually spam, what fraction is predicted as spam. Precision and recall are both high around 0.9, meaning that the spam prediction almost always correctly classifies the SMS.