UE19CS322
Big Data Project
Final Report

Team No. - BD_067_071_167_544

Team Members:
Anisha Ghosh - PES1UG19CS067
Anupama Nhavalore - PES1UG19CS071
Gayathri Sunil - PES1UG19CS167
Tushar Shetty - PES1UG19CS544

3.1 Machine Learning with Spark Streaming: Twitter Sentiment Analysis

3.2. Design Details :

Most of our design decisions for the models themselves were based on the fact that we had to implement incremental learning given the size of our dataset. We started our implementation off by streaming our dataset using Spark. We implemented and played around with streaming the sentiment dataset using different batch sizes, namely - 2000, 5000, 10000.

The stream.py file would ensure the streaming of the data at the localhost according to the batch size given as input. The stream would be collected at the client-side in order to be able to implement incremental learning. These batches of data would be collected by iterating through the DStream object and extracting each rdd for preprocessing and modelling.

The models we've implemented are:

1. Stochastic Gradient Descent(SGD)
2. Multinomial Naive Bayes Classifier(MNB)
3. Bernoulli Naive Bayes Classifier(BNB)
4. KMeans Clustering(KMC)

Before testing on test.csv, we parameter tuned the models chosen using a train-test split on the train.csv.

Please note, we attempted implementing a Multilayer Perceptron model, but due to low accuracy chose not to analyse the same. However, the model can still be found in our repository.

3.3. Surface Level implementation details about each unit :

Our project involves machine learning under 3 different models and one clustering algorithm. These 3 models include Multinomial Naive Bayes Classifier, Bernoulli Naive Bayes classifier and Stochastic Gradient Descent Classifier. Under clustering we've implemented incremental clustering with the k means clustering algorithm

The general methodology of every incremental learning model was implemented in a similar fashion as detailed below:

1. **Streaming**
   The batches of data coming in from stream.py would be collected on the client-side and given as input to our model.py. This Dstream object of RDD's was then read, manipulated and converted to a Pyspark data frame under the *def temp()* function. The model object is then instantiated globally once which is then used to train incrementally over batches of data. The final RDD extracted from the Dstream object is then sent to the preprocessing and training function.

2. **Preprocessing**
   The preprocess function *def preprocessing()* takes care of various things:

   - Firstly, with the help of the regex library, we are able to remove unwanted characters in the tweet including extra spaces letters appended for byte strings, and removing " 's " from strings.
   - We converted the strings to lowercase and performed lemmatization on it to get a better set of features for further preprocessing.
   - Using a predefined library of stopwords, we removed stopwords from the strings
   - Now we have introduced a Hashvectorizer() in order to introduce a metric to be able to train the data. This is a sklearn library that uses a hashing function to determine the frequency of items in the document along with vectorizing these items returning vectors for each document which is a good metric that can be used in training the model.

3. **Training**
   Now we use a sklearn method called *partial fit(X, Y, classes)*. This is the key to incremental learning models and is used to allow the retraining of the model over every batch that is passed as input to the code. This is done over the entire training set iteratively. Over every iteration, a pickle is used to dump the model file and over the next iteration, this model is loaded and further partially fitted until all the batches have been trained.

Hyperparameters for every model:

```
1. SGDClassifier(alpha=.0001, loss='log', penalty='l2', n_jobs=-1,
   shuffle=True)
2. MultinomialNB(alpha=1.0,fit_prior=True)
3. BernoulliNB()
4. MiniBatchKMeans(n_clusters=2,random_state=0,batch_size=10000)
5. MLPClassifier(activation='tanh',learning_rate='constant',alpha=1e-4,
   hidden_layer_sizes=(70,), random_state=1, batch_size=500,verbose=
   False, max_iter=1, warm_start=True,shuffle=True)
```

4. **Testing**

On obtaining this final incremental model the test data is now streamed to the localhost. This data is passed into the code onto the client-side. The model is now loaded and predicts the class value for the batch inputs coming in from the test dataset using *model.predict()*. These values after prediction are compared against the corresponding classes of these batches and the following performance metrics are obtained:

- Accuracy
- Precision
- Recall

We have visualised these in <section 3.5> to portray how the model is incrementally learning.

3.4. Reason behind design decisions :

Preprocessing functions and why we chose them:

- Regex Library:

  The *re library* was used as it helps deal with the basic, arbitrary preprocessing efficiently. This is used to take care of certain repeating unwanted characters within each document. We chose to remove some annotations using regex as it's efficient in the way it swaps the character with certain spaces.

- Choice Of Vectorizer:

  Hashing vectorizer was used over the usual Count Cectorizers because a hashing function is a more efficient way to map word counts over word counts. It produces a sparse matrix consisting of negative and non-negative values. These can be dealt with with the help of various parameters while initialising the class such as setting alternate signs as true/ false depending on the size of the sparse matrix produced. This proved to

be very helpful during the implementation of the Naive Bayes classifier as the size of the sparse matrix was very large due to the negative values vectors.

- NLTK libraries:
  Further, *nltk* has been used to import methods like *Stopwords* and *WordNetLlemmatizers* so that common words such as names, URLs, grammatical components such as conjunctions, articles e.t.c. do not contribute to the score of the tweet. Lemmatizers are used so that similar words such as participles of words contribute to the same base word and thus efficiently calculate the score.

- Clustering
  Unsupervised algorithms such as clustering, are difficult to implement incremental learning due to 2 main reasons:

  a. It doesn't go through a set training phase and starts classifying with every batch as it comes through. Given that the initial set of centroids is chosen at random and it's subject to incremental learning, it can be misleading and give poor results as seen in our analysis too. Our approach to dealing with this was choosing a random state of centroids based on the continuous flow of data on every iteration.

  b. KMeans clustering is very sensitive to outliers and upon encountering values like this in every batch, the centroids can change by a multitude misleading all the classification till that point and hence yielding poor results. Therefore such a model produces poor accuracy.

Amongst our choices for clustering algorithms, however, k-means is the best choice as it adjusts the data better incrementally compared to other clustering algorithms.

3.4. Takeaways from the project :

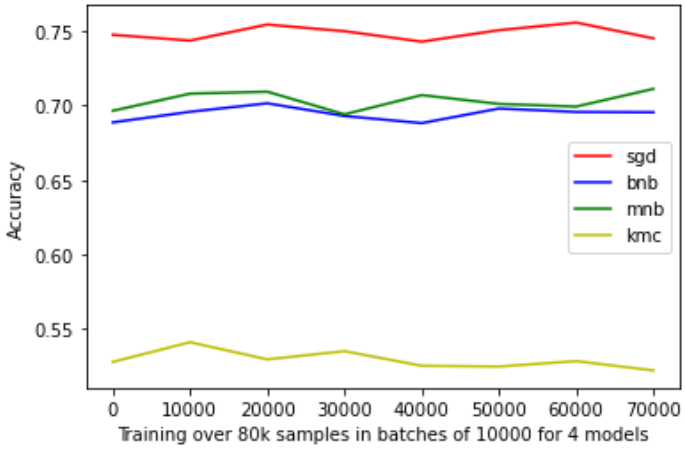
1. Ability to understand how streaming of data can be implemented on Spark

2. Understanding of how incremental learning can be implemented through streaming batches of data

3. Comparison of how each model under study performs under incremental learning:

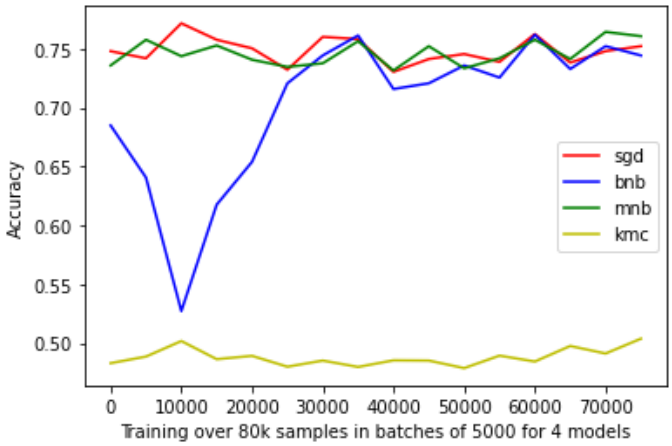
Comparison between performance metrics of all models based on batch sizes

a) Performance(Accuracy) At batch size 10000 :



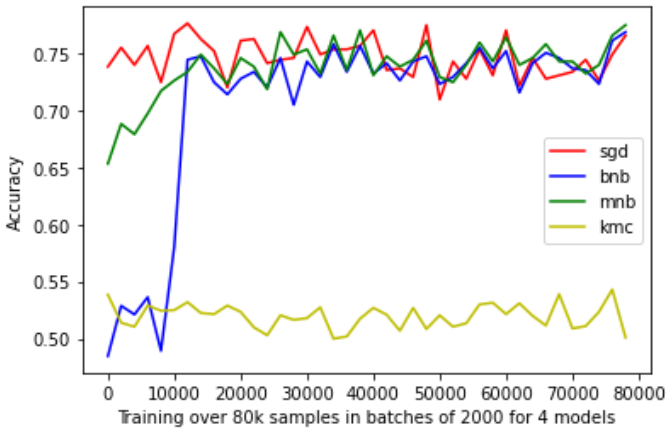
> SGD Model performs the best with a batch size of 10000, as observed KMC is the worst, clearly indicating that clustering is not a good method for sentiment analysis as it's an unsupervised model.

b) Performance(Accuracy) at batch size 5000 :



> The drop in BNB can be due to the fact that the model did not perform well for a particular test batch, this stabilizes towards the end.
> The accuracy of all the 3 supervised models are comparable for this batch size.

c) **Performance(Accuracy) at batch size 2000 :**



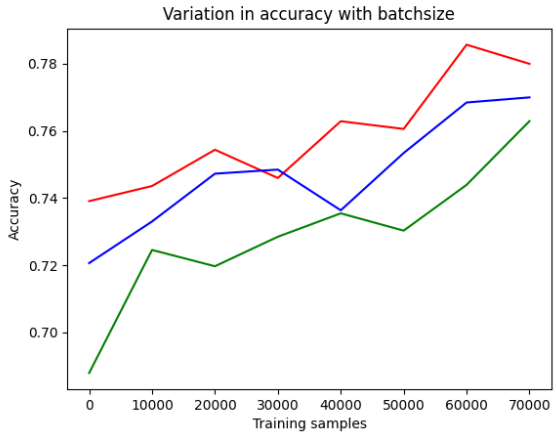
> For a batch size of 2000 the KMC's performance improves by a small value but reducing the batch size further would increase the time to train the model.
> Again, the accuracies of the 3 supervised models are comparable.

Overall, we see that across all batch sizes - SGD performs the best while KMC is the worst throughout.

Performance metrics of each model as compared to various batch sizes

a) **SGD**

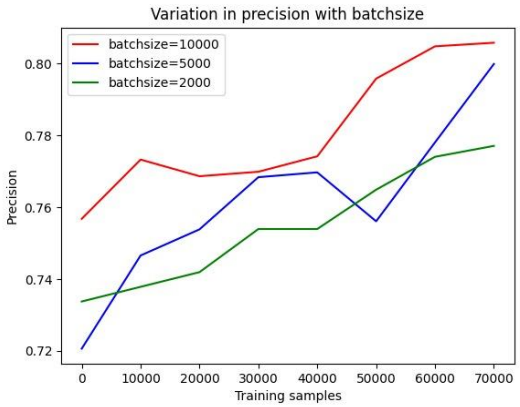
i) Accuracy



Red: 10000, blue: 5000, green: 2000

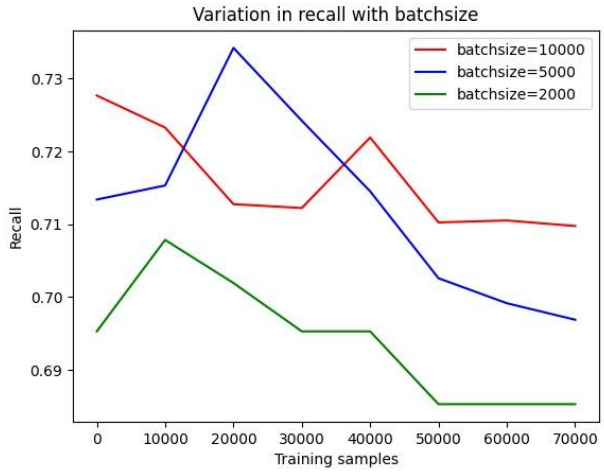
> In SGD, The accuracy of the model increases with increase in batch size

ii) Precision



> The precision is comparable to the accuracy plot and clearly as batch size increases the precision increases.

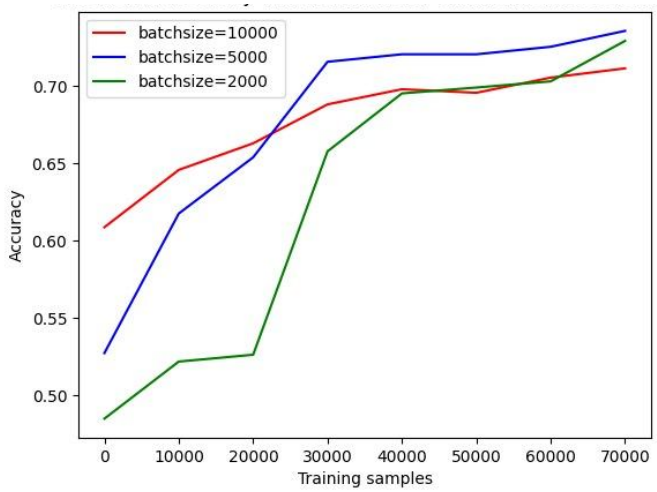
iii) Recall



> The recall eventually decreases in all the batches and they all have an average recall of 0.7 which is acceptable.

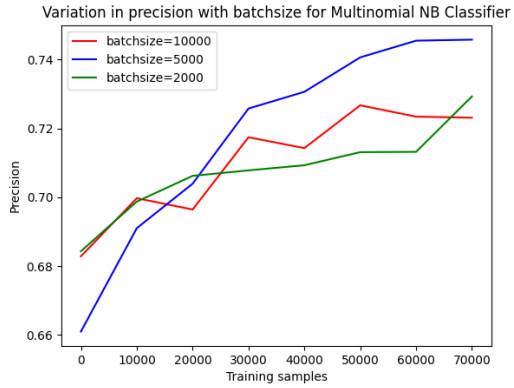
b) Bernoulli Naive Bayes Classifier

i)Accuracy



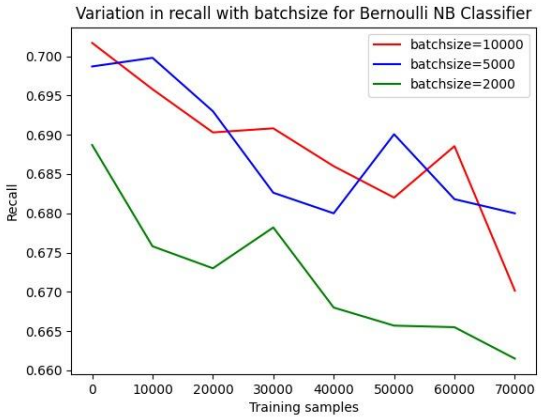
> The accuracy of the BNB Model, the models trained using 10000, 5000 are comparable with 5000 batch size performing slightly better than the former. The model trained using a batch size of 2000 is unstable as the accuracies aren't consistent.

ii) Precision



> The model trained using batch size 5000 has the best precision, therefore, it's the best model out of all the other models trained on different batch sizes.

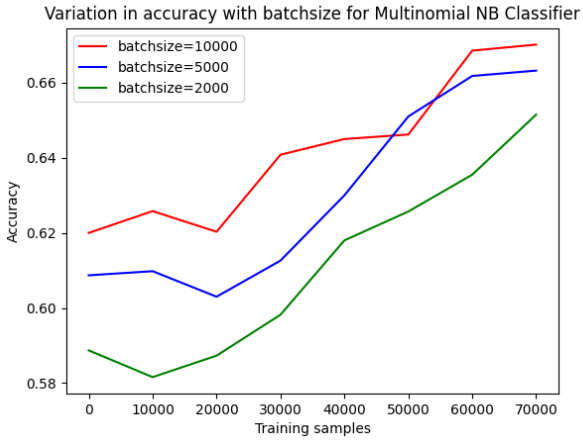
iii) Recall



> The model trained using batch size as 10000, 5000 have comparable recalls which is consistent with the accuracies and precision on the BNB model.

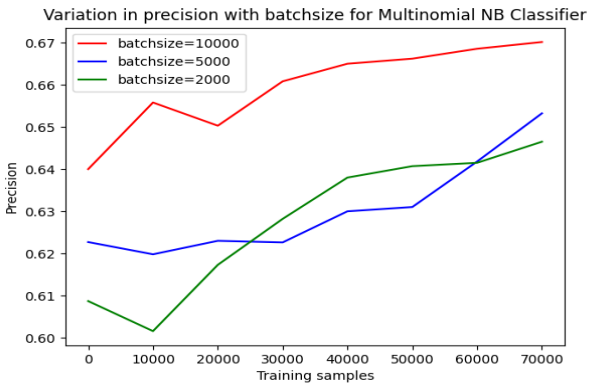
c) Multinomial Naive Bayes Classifier

i) Accuracy



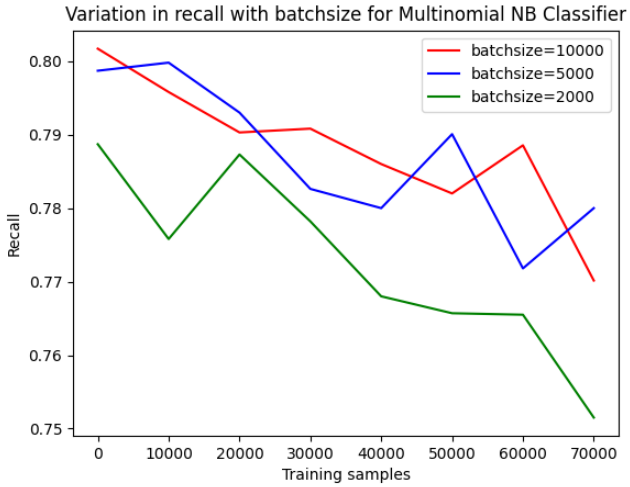
> The models trained using 10,000 and 5000 batch sizes have the highest accuracy indicating that increasing batch sizes improves accuracy.

ii) Precision



>model with batch size 10000 has best precision indicating it's the best model out of the 3 even though they have comparable accuracies.

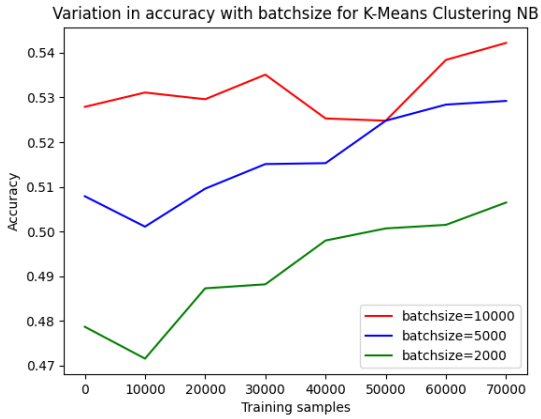
iii) Recall



> models with batch size 10000 and 5000 have comparable recalls, this is consistent with the precision and accuracy for the corresponding batch sizes.

i) K Means Clustering

iv) Accuracy



>model with batch size 10000 has the best accuracy but overall the accuracy is very low therefore this is not a good model for sentiment analysis.

Overall, we see that Stochastic Gradient Descent performs the best in terms of incremental learning while K-Means and clustering algorithms clearly don't adapt to iterative incremental learning for our analysis of sentiment.