Real-Time Twitter Hate Speech Detection

JiangLong He, Chen Liao, Kaining Mao, Mary Wu, Pavan Varma Bhupatiraju
jh8011, cl6108, km5745, bw2327, pb2612

Problem Statement

Social media has become a prevalent part of daily life, with many platforms for users to publicly express their thoughts and views. For instance, Twitter and Reddit have over 200 million [1] and 50 million [2] daily active users, respectively. As their popularity continues growing over the years, these social media platforms are under increased scrutiny to detect and filter out hate speech (amongst other things, such as misinformation). For instance, Twitter explicitly states in their policy that one of their core values is to represent all voices, including marginalized and underrepresented groups [3]. It is not evident whether they use hate detection models on their platforms, but they do encourage users to report any hateful behavior.

For our project, we develop a real-time system to detect hate speech from Twitter. The end goal of the project is to be able to either filter out or at least flag hateful tweets from Twitter in real time, to align with their mission to prevent hateful conduct. There are potentially half a billion tweets on a given day [4], so our solution requires the use of big data tools. We implemented the pipeline to detect hate speech in real time, using MongoDB, Kafka, Pyspark, and AWS EC2 instances.

The rest of the report is structured as follows. We first begin with an explanation as to why we need to use big data tools for this problem statement. We then present a high-level overview of our design, followed by the details of the individual components. We present the key findings, and then conclude with potential future work.

Why Big Data?

Statistics show that there are roughly 500 million tweets per day [5], out of which there are roughly 1,300 hate tweets reported per day, rising to almost 4,000 after Elon Musk took over Twitter [6]. We are looking at 1/129,000 odds of detecting a hate tweet. Without automation this wouldn’t be feasible. Let's assume an 8 hrs shift per laborer, 2 tweets/sec, and a minimum wage of $12/hr. A laborer can manually check 57,600 tweets per day (without any breaks). If we are aiming to check ⅕ of the total daily tweets, we are looking at 100M tweets to process. That would cost Twitter $166,000 per day in wages, a costly endeavor.

As mentioned earlier, there are upwards of half a billion tweets being posted per day around the world. This is an extremely large number of tweets to process and analyze, especially in real time. Given the sheer volume of data that needs to be processed in our use case, this cannot be handled on a regular server, using tools that are not specifically designed to handle large data.

This problem requires a database to store the data (e.g. MongoDB) as well as a way to manage the database to ensure it does not explode in volume (e.g. aggregate when possible and remove historical data that are not used in visualizations). Our project also requires the use of tools, such as Pyspark, to perform data transformations (e.g. preprocessing of the tweets prior to the model prediction). Traditional methods, such as Pandas, would be much slower and might not even be able to store the data in local memory to perform the necessary operations.

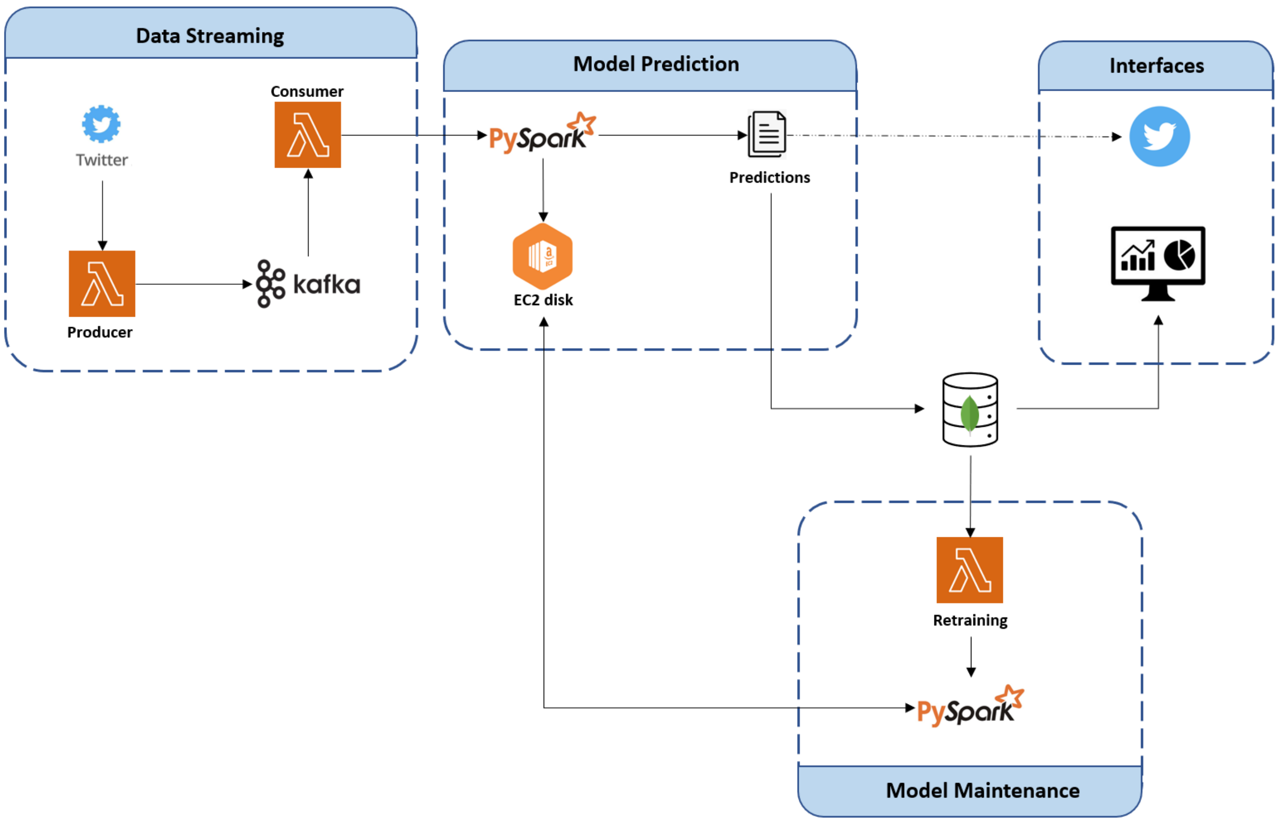
High-Level Design Overview

Training Phase

The first portion of our project consisted of conducting an exploratory data analysis (EDA) of the training data and training the initial prediction model. We downloaded a dataset from Kaggle consisting of roughly 50,000 samples of labeled tweets (binary labels, with 0=not-hate, and 1=hate) [4]. Through the EDA, we gathered insights into the distribution of the data, which guided our model training phase. Once we trained an adequate model, we moved on to creating a real-time production pipeline that would support real-time prediction and model maintenance.

Production Phase

The production pipeline we designed is shown below. The following high-level overview shows that we are able to stream real-time data, generate model predictions, and render the results on a dashboard for our user to see (for instance, if our client is Twitter, this would allow their employees to monitor the results of our prediction model). There is also a dotted-line pointed to the Twitter interface, suggesting a potential future step to integrate our model predictions into Twitter itself, to filter and/or flag hateful tweets.



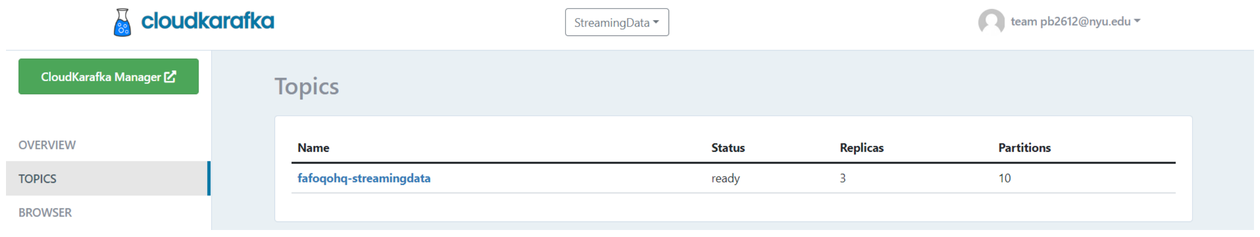
Design Details

This section will dive into the specifics behind each of the components in our pipeline. All four modules in the architecture are hosted using cloud services, so the entire pipeline is online and triggered without manual intervention.

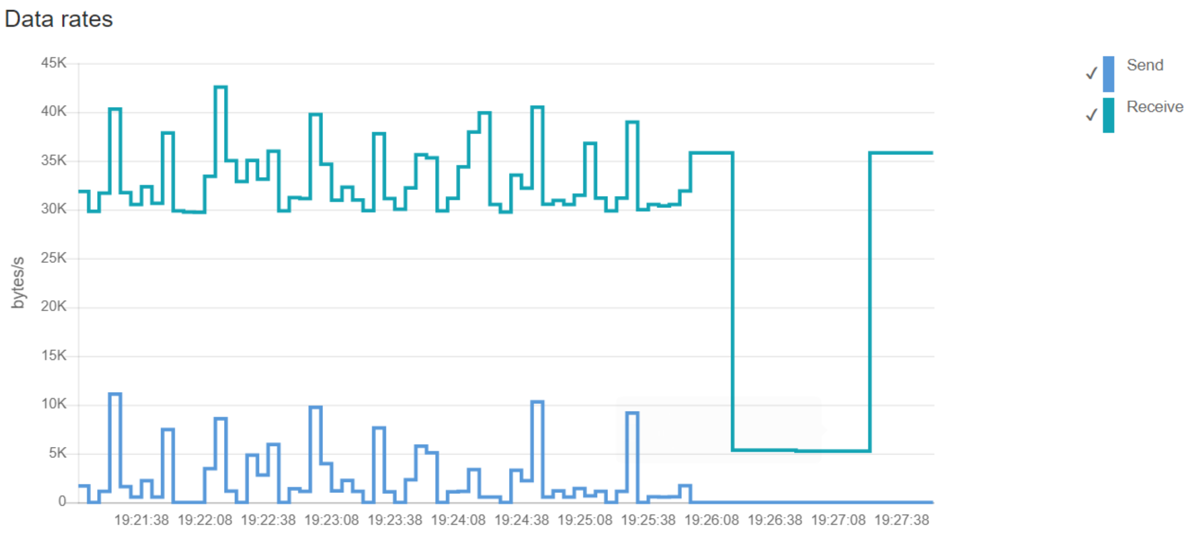
Data Streaming Phase

In the data streaming phase we are using Apache Kafka to stream tweets in real time, as it allows for the ingestion, storage, and processing of large volumes of data. First, the stream of tweets from the Twitter API are ingested into Apache Kafka using a custom producer python script. The Twitter API with Essential access has a monthly cap of 500k tweets, so the producer script also has a limit with a cap for each run. Each tweet information is pushed as a single json record in kafka containing tweet id, tweet text and created time stamp. As the order of retrieving the tweets is not relevant here the records are pushed to Kafka without providing the partition key, so the data is automatically balanced in the 10 partitions created for the topic.

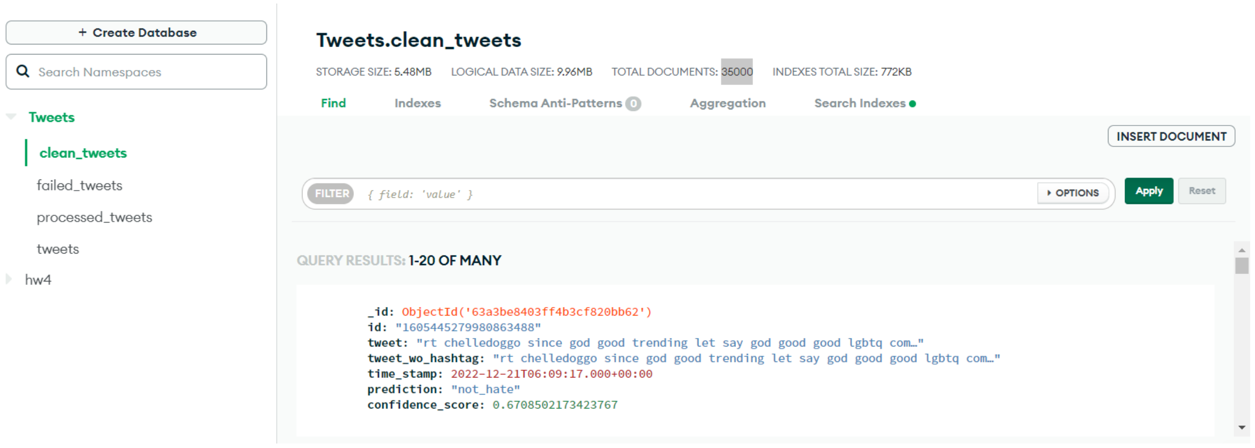
System Configurations:
Our Kafka clusters are managed by cloudkarafka. For our project, we configured Apache Kafka with three brokers. A single topic is created with 10 partitions for streaming Twitter data.



Name	Status	Replicas	Partitions
fafoqohq-streamingdata	ready	3	10



The producer script is scheduled to run every 15 minutes and for each run it pushes 2000-2100 tweets to the Kafka cluster. The consumer script is also triggered every 10 minutes to get the new tweets from Kafka clusters and push them to the model prediction phase. We used AWS EC2 for shared memory for model features storing and also to timely trigger the Producer, Consumer scripts using Cron scheduler. To date, we processed 35000 tweets in real time using the current system.



Tweets.clean_tweets

STORAGE SIZE: 5.48MB LOGICAL DATA SIZE: 9.96MB TOTAL DOCUMENTS: 35000 INDEXES TOTAL SIZE: 772KB

```
{ field: 'value' }
```

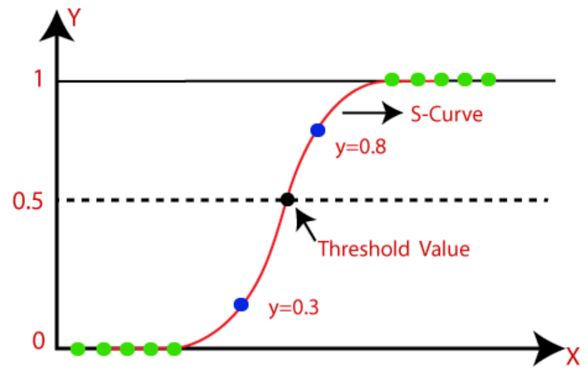
QUERY RESULTS: 1-20 OF MANY

```
_id: ObjectId('63a3be8403ff4b3cf820bb62')
id: "1605445279980863488"
tweet: "rt chelledoggo since god good trending let say god good good lgbtq com…"
tweet_wo_hashtag: "rt chelledoggo since god good trending let say god good good lgbtq com…"
time_stamp: 2022-12-21T06:09:17.000+00:00
prediction: "not_hate"
confidence_score: 0.6708502173423767
```

Model Training Phase

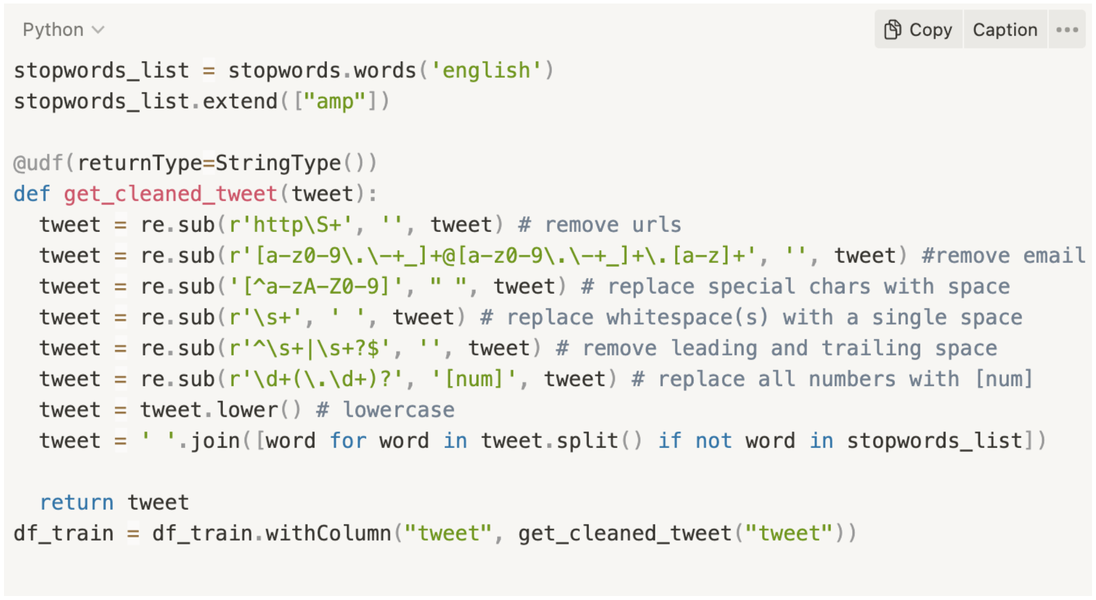
Model of Choice (Logistic Regression)

Given the initial labeled data we downloaded from Kaggle consisting of roughly 50,000 samples of labeled tweets (binary labels, with 0=not-hate, and 1=hate) [4], we can start training our initial hate tweet classification model. We have chosen a simple Logistic Regression as our classifier. Logistic regression outputs a probability score when given a set of features as input. The probability score ranges from [0, 1]. To achieve binary classification we can simply set a threshold value, any probability higher than the threshold we will consider it a prediction of class 1, in our case 1 = hate speech. We have chosen a threshold of 0.5.



Data Preprocessing

We perform some data preprocess using pyspark before using the data to train our model. We first drop duplicate rows such that both tweet and label are equal to each other using *dropDuplicates()* method. A total count of 3,130 rows were dropped during this process. Next clean off each tweets by using an *udf* function:



```python
stopwords_list = stopwords.words('english')
stopwords_list.extend(["amp"])

@udf(returnType=StringType())
def get_cleaned_tweet(tweet):
  tweet = re.sub(r'http\S+', '', tweet) # remove urls
  tweet = re.sub(r'[a-z0-9\.\-+_]+@[a-z0-9\.\-+_]+\.[a-z]+', '', tweet) #remove email
  tweet = re.sub('[^a-zA-Z0-9]', " ", tweet) # replace special chars with space
  tweet = re.sub(r'\s+', ' ', tweet) # replace whitespace(s) with a single space
  tweet = re.sub(r'^\s+|\s+?$', '', tweet) # remove leading and trailing space
  tweet = re.sub(r'\d+(\.\d+)?', '[num]', tweet) # replace all numbers with [num]
  tweet = tweet.lower() # lowercase
  tweet = ' '.join([word for word in tweet.split() if not word in stopwords_list])

  return tweet
df_train = df_train.withColumn("tweet", get_cleaned_tweet("tweet"))
```

Above *udf* function will remove urls and emails. It will remove special characters from each word and remove all unnecessary spaces. We also chose to lowercase our words so that number of vocabulary can be minimized. We have also replaced all numerical words with a special token *[num]* to indicate that it is a number. This allows us to reduce the number of unique word count by a huge margin. Last but not least we remove any stopwords that don't carry much semantic meaning. We ended up with 250,447 words and 37,927 unique words.

Data Transformation

Before we feed the data to the model, we need to transform the preprocess tweet into numerical features. We have decided to use the *TF-IDF (term frequency - inverse term frequency)* feature as an input feature to our Logistic Regression model. Reason being that during the initial EDA phase we found that tweets that were labeled as Hate-speech carries a lot of similar words such as: trump, race, white, and etc. Knowing that there are a specific set of keywords in hate tweets, we can capture the signal using the IDF feature which computes how common or rare a word is in the entire document set. The closer it is to 0, the more common or rare a word is in the entire document set. I would expect these keywords will result in IDF value closer to 1 since only 7% of tweets are marked as hate. We leverage the Pyspark mllib library to perform these data transformations.

```python
from pyspark.ml.feature import HashingTF, IDF, Tokenizer

# tokenization: split sentence into words
tokenizer = Tokenizer(inputCol="tweet", outputCol="words")
wordsData = tokenizer.transform(df_train)

# term frequency
hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures")
featurizedData = hashingTF.transform(wordsData)

# inverse document frequency
idf = IDF(inputCol="rawFeatures", outputCol="features")
idfModel = idf.fit(featurizedData)
rescaledData = idfModel.transform(featurizedData)
```

We added an extra transformation to compute the class weight. Since we have a data imbalance issue (7% hate), it can be helpful to assign different class weights to the model while training so that it does overfit on the majority class. We compute the class weight for hate-speech by doing $1 - count(hate)/total$ resulting in a weight of 0.9318 for hate class and 0.06817 for non-hate.

Model Training

After preparing the data we can start training the model. We performed a random 8:2 split to create a training and evaluation dataset. We performed hyperparameter tuning on following set of hyperparameters:

```python
# hyperparamters
regParams = [0.001, 0.003, 0.01, 0.03, 0.1, 0.3]
elasticNetParams = [0.001, 0.003, 0.01, 0.015, 0.02]
families = ["binomial", "multinomial"]
weightCols = ["no_weight", "weight"]
```

regParams is regularization parameter for L1 penalty. elasticNetParams is the ElasticNet mixing parameter in range [0,1]. For value 0, penalty is L2 penalty, for value of 1, it will be L1. families is the label distribution to be used in the model. Last but not least the wightCols will be to use class weights that we have precomputed on model training or not. We first perform an initial training with no class weight, no regularization, all other parameters set to default and obtain the following metric on the evaluation set. The model heavily overfits the minority class. We got an F1-score of 0.08 for class=Hate and 0.967 for class=non-hate. We can see the class imbalance issue hurts.

```
Confusion Matrix:
      Pred=0   Pred=1
True=0: [5526.    0.]
True=1: [ 376.   17.]

Overall F1-score: 0.9083925921908943
Overall Recall: 0.9364757560398715
Overall Precision: 0.9405227088912793

Class=1(Hate) F1-score: 0.08292682926829269
Class=1(Hate) Recall: 0.043256997455470736
Class=1(Hate) Precision: 1.0

Class=0(Not Hate) F1-score: 0.9670983549177459
Class=0(Not Hate) Recall: 1.0
Class=0(Not Hate) Precision: 0.9362927821077601
```

After performing hyperparameter tuning we are able to reach a F1 score of 0.64 for class=Hate and still maintain a high F1 score of 0.97 for class=non-hate.

```
# Best Hyperparameters:
regParam = 0.1
weightCol = "weight"
family = "binomial"
elasticNetParam = 0.015

# metric
Confusion Matrix:
      Pred=0   Pred=1
True=0: [5377.  149.]
True=1: [ 141.  252.]

Overall F1-score: 0.9512343170756914
Overall Recall: 0.9510052373711776
Overall Precision: 0.9514729087154187

Class=1(Hate) F1-score: 0.6347607052896725
Class=1(Hate) Recall: 0.6412213740458015
Class=1(Hate) Precision: 0.628428927680798

Class=0(Not Hate) F1-score: 0.9737413980441868
Class=0(Not Hate) Recall: 0.9730365544697792
Class=0(Not Hate) Precision: 0.9744472635012685
```

Model Prediction Phase

During the model prediction phase, we have deployed our trained pyspark model onto an EC2 instance where a spark environment is installed. The trained model is stored on the shared memory in EC2 instance. Model prediction will be triggered when the KafkaConsumer script is invoked. The script will first fetch the tweet data from Kafka queue. We then generate a pyspark dataframe from the data. Then the same pipeline of data preprocessing and transformation will be performed on the dataframe to obtain input features. The script will then load the trained Logistic Regression model from EC2 and perform inference to obtain the prediction.

Since we are using a logistic regression model, we get a probability score as the prediction result. We can utilize the probability by treating it as a confidence score to detect uncertain predictions performed by model. We set a confidence score threshold of 0.6. So any prediction with lower than 0.6 confidence score will be gathered and pushed to our MongoDB *fail_tweets* collection for model maintenance phase. Rest of the predictions will be stored in both *processed_tweets* and *clean_tweets* collections for data visualization purposes.

Model Maintenance

Model maintenance phase is designed to keep our model updated to adapt to data drift. The uncertain predictions from **Model Prediction Phase** stored in MongoDB will be manually annotated by human annotators. To set up the system, we use random generated labels to test the model retraining. However, once the system is set up, with the help of good annotation quality, we can easily update our model to adapt to the drift of the data stream. We will trigger model retraining every 24 hours to retrain our model with the new data points. The updated model will overwrite the old model file in EC2 shared memory and be ready for any further prediction jobs.

MongoDB

Due to the various data formats we would handle during the project, MongoDB Atlas, as a NoSQL database, was applied for full data support. It provides a remote elastic database as a service, increasing accessibility, scalability, and modularity of our system. MongoDB Altas also provides advanced monitoring of collections, with MongoChart providing real-time synchronized visualizations, which ease our deployment procedure. There are three collections in our database: clean_tweets, processed_tweets, and failed_tweets. The processed_tweets collection stores individual words or phrases as a document with its meta data. This collection serves for our visualization of hate word frequency ranking. Entries are requested from this collection and grouped by their time stamp. In this way we could count word frequencies based on the number of documents in each aggregation in a single day.

The clean_tweets collection stores sanitized tweets with its metadata. The consumer model triggers the fetch from our Kafka message queue every 15 minutes. Data will pass through preprocessing scripts for sanitation, which will subsequently be stored in this collection. This collection mainly serves for general purpose visualizations, including aspects such as hate vs non-hate and the distribution of hate confidence scores.

The failed_tweets collection contains tweets that are not confidently classified by our model. As mentioned, the model will assign each tweet a confidence score, indicating how confident the model is about the classification result of this specific tweet. Taking a threshold of 0.6, we regard tweets with confidence score less than it as failed tweets, which will be stored in this collection for further retraining. Therefore, the schema of this collection only contains the tweet itself and its manually assigned label, which avoids processing phase after retrieval and thus simplifies the retraining part.

Insights

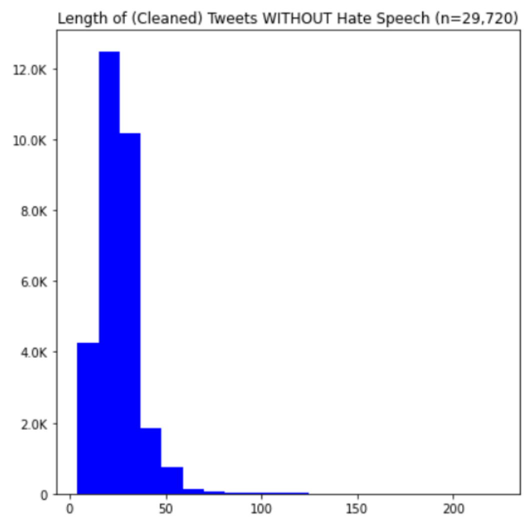
We will discuss insights from both phases of the project. From the training phase, we conducted an exploratory data analysis (EDA) to gain insights from the training data prior to building the predictive model. After we created the production pipeline, we also generate visualizations in real time, from the data we stream from Twitter. Naturally, the visualizations are slightly different in the production phase, so we report on those as well.

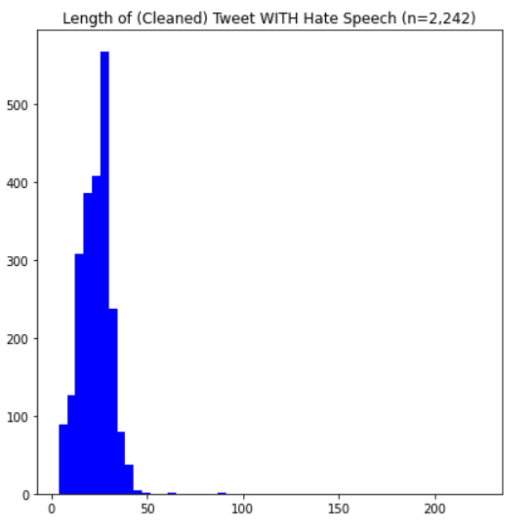
Exploratory Data Analysis

One of the first steps of the project was to conduct an EDA on a training dataset, to better understand the structure and trends of the dataset. From our training dataset of 40k tweets, we looked at class distribution, distribution of Tweet data, detection of outliers, and top words used in hate speech versus regular speech.

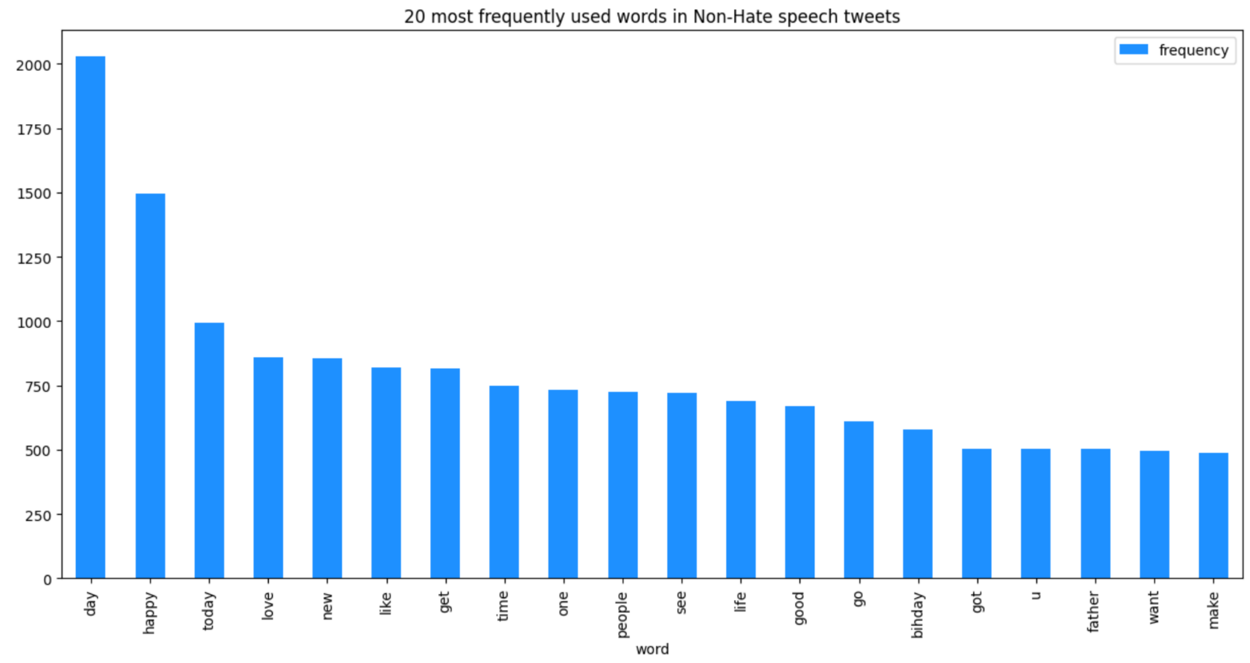
The first observation we noted was the class imbalance; only roughly 7% of the dataset was labeled as hate speech. This was expected but interesting to note, because this suggests two things. First, we remark that hate speech represents the minority of tweets on Twitter, which may present as a challenge during the model training phase. It is well-known that training on imbalanced classes may result in suboptimal models that neglect to detect trends from the minority class. This could result in high levels of false negatives, rendering the model useless in production. There are many ways to rectify this problem in the literature, but for the scope of our project, we only flagged this as a potential improvement and went ahead with our model. The second remark is that Twitter may not have a pre-existing algorithm to filter out hate speech, providing all the more motivation for our project.

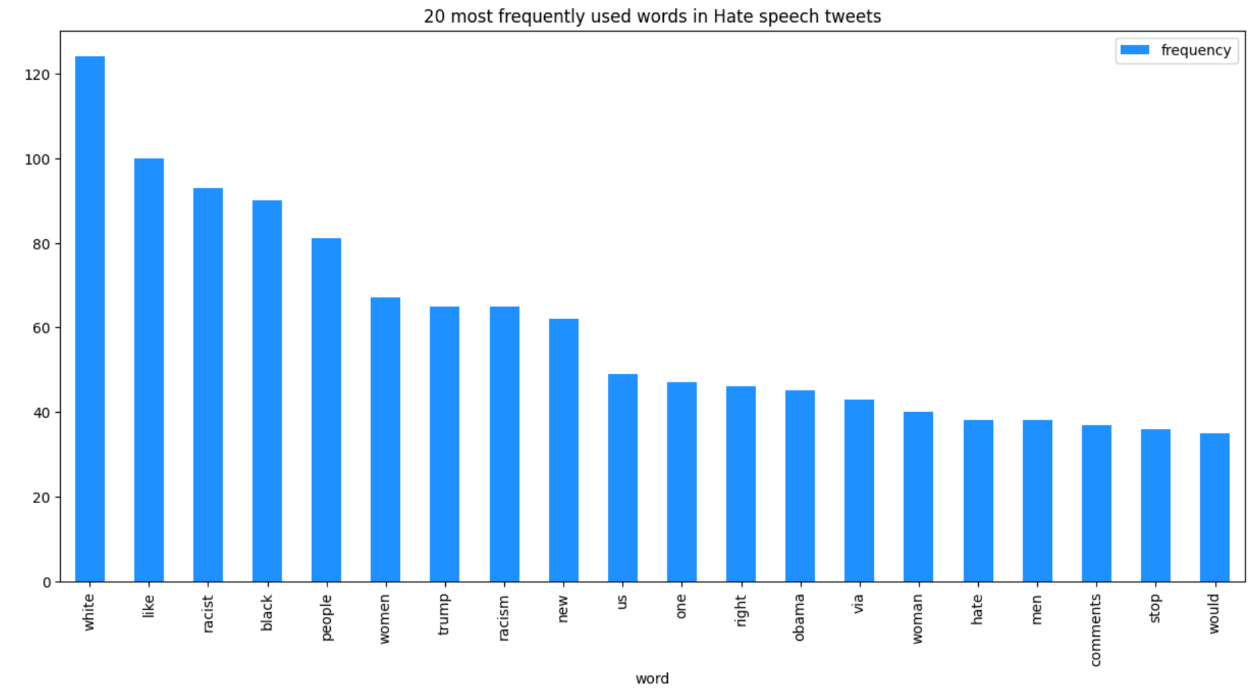
The second part of our EDA was to investigate the size of the tweets (number of words). Twitter has a 300 character limit, which creates a small range of possible word lengths. This is beneficial for model training, since we do not have to worry about dealing with sequences that are significantly shorter or longer than others. There were some that were shorter (i.e. only a few words long), but they were rare in our dataset. As such, there was not much concern in regards to tweet length, and we proceeded to the next step.





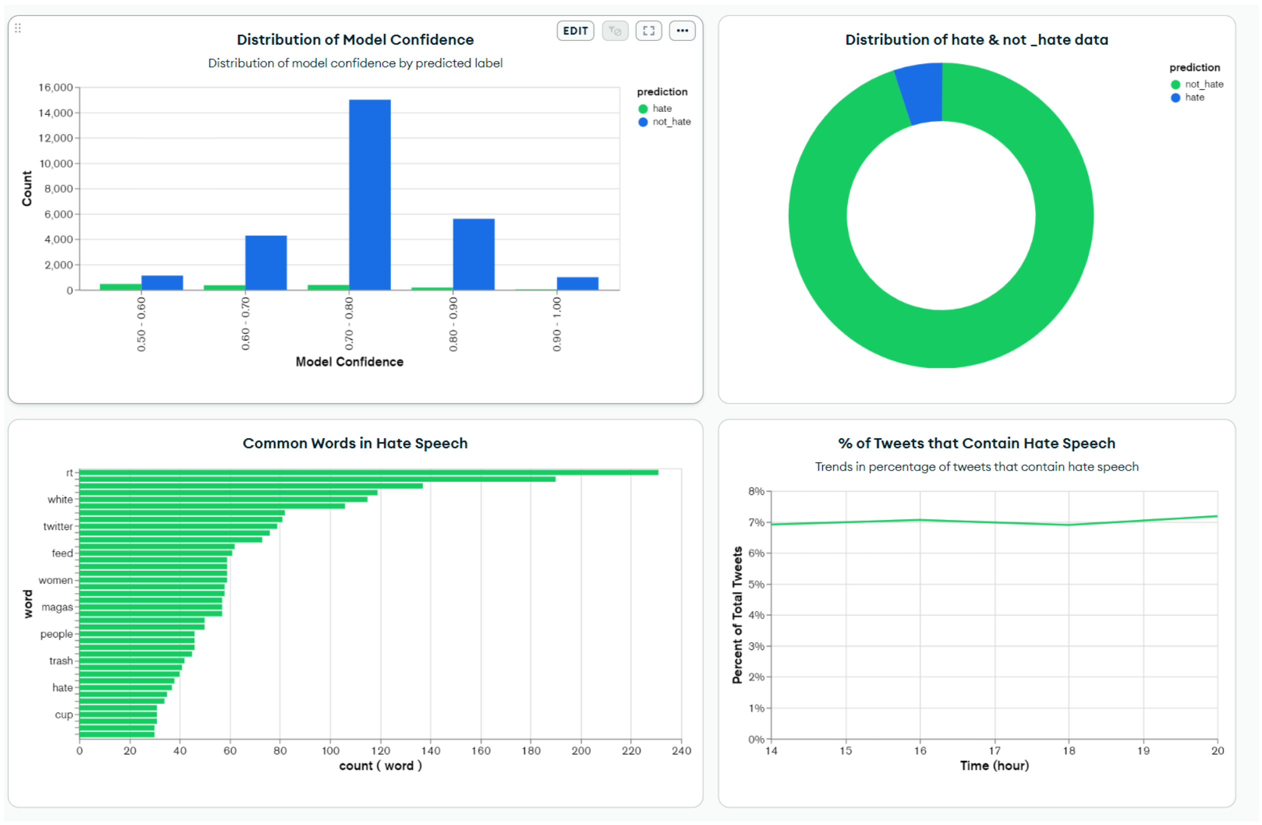
The next part of the EDA was to visualize the top words used in each of the classes (hate speech and regular speech). This would allow us to see what words are common and reappearing, especially amongst the hate speech tweets. In our analysis, we found that the top words were extremely different across the two classes. In the hate speech group, there were some words related to politicians, words with angry connotations, and such. In the regular speech group, most words were neutral, but there were even some positive words. This was really interesting to observe, as well as reassuring, since this suggests that there are in fact differences in the two classes, so the model should hopefully be able to detect this as well.





Data Visualizations in Production

To visualize the results in real-time in our production pipeline, we used MongoDB Charts. Since our data was stored in MongoDB Atlas, MongoDB Charts was an easily compatible choice. Another important factor in our decision is the fact that MongoDB Charts refreshes the visualizations automatically, ensuring that the dashboard visualizations are in line with the database updates. This is extremely useful for our users to monitor the real-time results (without having to manually refresh). A screenshot of the dashboard is shown below, followed by descriptions of each of the four visuals.



Distribution of Model Confidence
Distribution of model confidence by predicted label
EDIT
Count
Model Confidence
0.50 - 0.60
0.60 - 0.70
0.70 - 0.80
0.80 - 0.90
0.90 - 1.00
prediction
hate
not_hate
Distribution of hate & not _hate data
prediction
not_hate
hate
Common Words in Hate Speech
word
rt
white
twitter
feed
women
magas
people
trash
hate
cup
count (word)
% of Tweets that Contain Hate Speech
Trends in percentage of tweets that contain hate speech
Percent of Total Tweets
Time (hour)

Distribution of Model Confidence:
Our prediction model outputs the prediction confidence from 0 to 1. We want to visualize the distribution of the model confidence to ensure that the overall model confidence rate is reasonable. Separating the histogram by prediction label also helps us identify if we are struggling to predict on the minority class, which seems to be the case here.

% of Tweets that Contain Hate Speech
The next graph we show is the fraction of tweets that are flagged as hate speech throughout the day. In the future, this could be extended to a larger time scale to show the percent of hate speech over the last week or month. The purpose of this graph is to identify if there are any peaks in hate speech (for instance, a rise in hate speech after significant political events). Another instance is before and after Elon Musk took over Twitter. It was observed that there was a significant peak in hate speech after the transition. These would be interesting for our client (Twitter) to note.

In our figure, we only had data for one day, so we visualized the percent of hate speech over the course of that day. The rate of hate speech seemed to be relatively stable throughout the day, suggesting there were no provoking events during that time.

Snapshot of Distribution of Hate Speech
Whereas the line chart above shows the trends over time, this graph serves as a snapshot to give us the distribution of hate speech from our most recent data batch. The purpose of this graph allows the user to quickly gauge the rate of hate speech on a particular day.

Common Words in Hate Speech
The fourth figure we generate is the word count of the most frequently used words in the tweets that were flagged as hate speech. This could be useful for our client to quickly gauge which words are currently trending in hateful tweets. This could be an indicator of what events are happening that are triggering hate speech; it may be useful for our client to be aware of such trends to place increased attention to the tweets containing flagged hate words.

Conclusion

In conclusion, we designed an end-to-end pipeline for real-time hate speech detection of Twitter data. We began with a training phase, which consisted of an exploratory data analysis of Kaggle data (consisting of ~50k labeled samples) to inform our model training. After developing the prediction model, we integrated it into a production pipeline which uses Kafka to stream real-time from Twitter, Pyspark to transform and perform the model inference, and finally, MongoDB to store the data. We then use MongoDB Charts to visualize the results of our model on a dashboard which refreshes automatically at every hour.

Through this project, we practiced using big data tools and learned to think about edge cases that may arise when handling large amounts of data. We also learned to set up a real-time streaming system using cloud services, another valuable tool when processing big data.

There are a few potential improvements we considered for the future. First, there is definitely potential to further improve the model. Given that this course was not focused on model development, we did not spend too much time on model creation. If this were to be implemented in Twitter, the model would need to have a much higher confidence, likely using much more sophisticated prediction models. Another potential improvement would be to create a more interactive UI that can scale to any user. For instance, it might be useful if the user could directly pull up the tweets that were flagged as hate on the interface directly.

Overall, this project was a very valuable learning experience, pushing us to consider different cases to ensure our product was scalable. We worked with many tools that were new to us, and we are excited to present a final product that is fully functional end-to-end. We are aware that there are still improvements to be made for this product to scale to worldwide Twitter data, but we believe that our deliverable is headed in the right direction.

Our code can be accessed at: https://github.com/Kylin-Mao/BD-HateSpeech.

References

[1] https://www.businessofapps.com/data/twitter-statistics/
[2] https://www.businessofapps.com/data/reddit-statistics/
[3] https://help.twitter.com/en/rules-and-policies/hateful-conduct-policy
[4]
https://www.kaggle.com/datasets/arkhoshghalb/twitter-sentiment-analysis-hatred-speech?select=test.csv
[5] https://www.internetlivestats.com/twitter-statistics/
[6] https://www.nytimes.com/2022/12/02/technology/twitter-hate-speech.html