Massive Data Analytics

Project

Ilia Hashemi Rad

99102456

News Data:

Explanations:

Since we have done most of the analysis of the news event in the first exercise, so I did a smart analysis on the data and processed an NLP algorithm for it, and I reached an acceptable result that we will see later.

New Data Analysis

Explanations:

In this analysis, I first separated the news that had the `ner_tag` field and performed an interesting analysis on the data based on their entity names. In this analysis, I first separated them based on the date of each news and then obtained the most frequent name entity of each day. This work was done in two steps, once for the name entities themselves: in which I obtained a list of the number of days of news and Sorted by time, it shows which name entity has been repeated the most in the news every day.

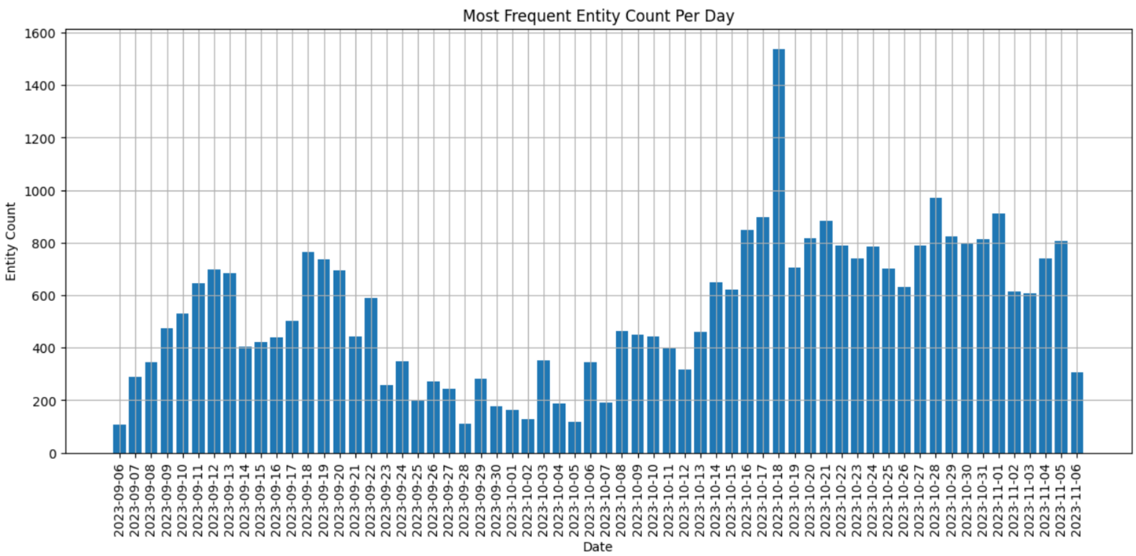
You can see part of its output here:

```python
print(sorted_entities_per_day)
```

```
[('2023-09-06', 'ایران'), ('2023-09-07', 'ایران'), ('2023-09-08', 'ایران'), ('2023-09-09', 'ایران'), ('2023-09-10', 'ایران'), ('2023-09-11', 'ایران'), ('2023-09-12', 'ایران'), ('2023-09-13', 'ایران'), ('2023-09-14', 'ایران'), ('2023-09-15
```

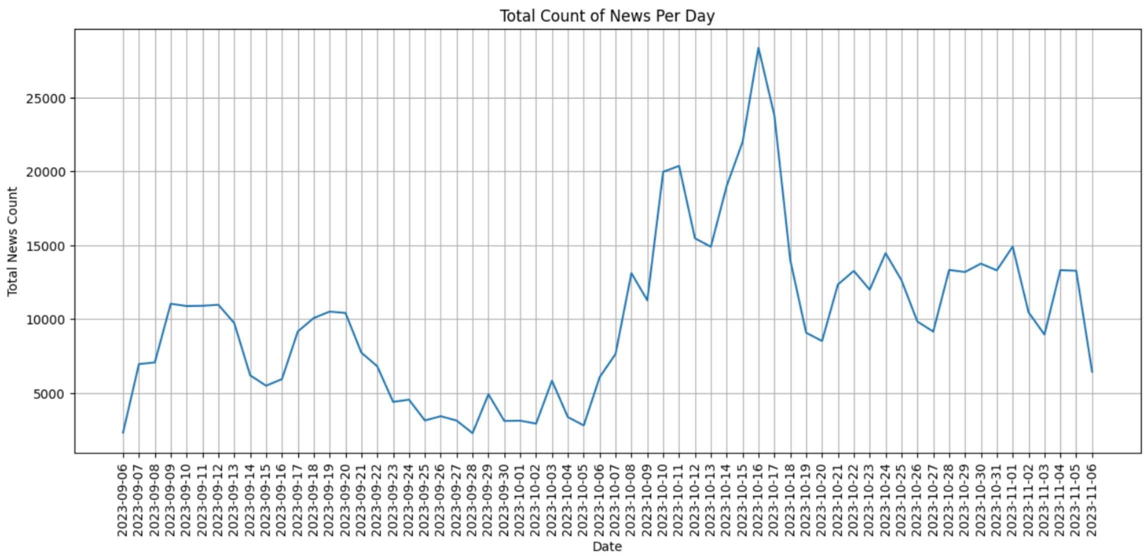
Explanations:

Then, in the next step, based on the date and the number of name entities, I drew them as shown below, where each bar corresponds to a day and shows the number of repetitions of the most frequent name entity of that day:



Explanations:

From the analysis of this graph, it is easy to justify the graph obtained in the first exercise, which was related to the timeline of the number of news. The chart mentioned in the first exercise was as follows. By comparing these two graphs, we can easily find out that when a particular peak or jump is observed in the number of news, the same thing happens to the most frequent name entity and the general shape of the two graphs is the same. Therefore, by examining that name entity, we can easily understand the reason for the jump in the number of news and understand what happened and what the news is focused on, for example, Gaza, as in the first exercise, we guessed that it is related to the operation. It was Al-Aqsa storm, here we were able to justify it with certainty



Classification of News

Explanations:

In this section, if I want to explain in general, I first filtered the news that has the categories field and cleaned and preprocessed their news text and stored it next to their category, which is their label, and then using the Machine Learning functions of the PySpark, I processed them and obtained the appropriate features of the news and finally trained a multiclass linear regression model so that it can recognize their category using the text of the news. Then, by testing the model, I reached 64% accuracy, which is very good accuracy for this amount of news and their long text and the fact that the processes were done on unique news and duplicated news had been removed, as well as the number of categories, which are ten. Also, the output is given as a probability distribution and is not just a label

```
categories = dedup_rdd.filter(lambda x: 'categories' in x and x['categories']).map(lambda x: x['categories'][0]).distinct().collect

print(f"The unique categories are : {categories}")
```

Explanations:

Here, is a brief explanations of my code:

- **Extracting Unique Categories:**
  - Filtered the `dedup_rdd`(which is the unique news without their duplicates) to include only records where the 'categories' key is present and non-empty.
  - Mapped each record to its first 'categories' entry.
  - Retrieved distinct categories using `distinct()`.
  - Collected the unique categories into the `categories` variable.

```
The unique categories are : ['health', 'sports', 'religious', 'politics', 'science_and_technology', 'economy', 'culture', 'security', 'social', 'military']
```

Explanations:

As you can see, we have 10 different classes which makes the NLP algorithm so hard to classify the news exactly as they are.

```
import re

# Define a function to remove useless characters including '\n', '\u200c', and '\\n'
def remove_useless_characters(text):
    # Remove special characters, digits, and unwanted unicode characters
    text = re.sub(r'\\|\\n|\\u200c|\s]', ' ', text)
```

```python
    # Define a regular expression pattern that matches one or more spaces
    pattern = re.compile(r" +")
    # Apply the pattern to the text and replace the matches with a single space
    text = pattern.sub(" ", text)
    return text

# Define a list of Farsi stop words
def load_stop_words(file_paths):
    stop_words_set = set()
    for file_path in file_paths:
        with open(file_path, 'r', encoding='utf-8') as file:
            stop_words_set.update(file.read().splitlines())
    return stop_words_set


# Apply the function to the news_rdd
title_category_rdd = dedup_rdd.filter(lambda x: 'categories' in x and 'body' in x and x['categories']).map(lambda x: (x['body'], x[

title_category_rdd = title_category_rdd.map(lambda x: (remove_useless_characters(x[0]), x[1]))

# List of paths to your stop words files
stop_words_files = ['verbal.txt', 'persian.txt', 'short.txt', 'chars.txt', 'nonverbal.txt']

# Load stop words from files
stop_words = load_stop_words(stop_words_files)

title_category_rdd = title_category_rdd.map(lambda x: (" ".join(filter(lambda w: w not in stop_words, x[0].split())), x[1]))

title_category_rdd.take(3)
```

Explanations:

Here, is a brief explanations of my code:

- **Text Preprocessing for News Categories:**
  - Defined a function, `remove_useless_characters`, to remove special characters, digits, and unwanted Unicode characters from a given text. Also, applied regex patterns to replace multiple spaces with a single space.
  - Defined a function, `load_stop_words`, to load Farsi stop words from specified files.
  - Applied the `remove_useless_characters` function to `dedup_rdd`, filtered relevant records, and mapped each record to a tuple containing the cleaned body text and its corresponding category.
  - Applied further processing steps:
    - Removed stop words from the cleaned body text.
    - Mapped the result to tuples containing the processed text and its corresponding category.
  - Displayed the first three processed records.

```
[('ی مسلح حضور فرمانده قوا برگزار مراسم مشترک دانشآموختگی دانشجویان دانشگاههای افسری نیروهای مسلح صبح سهشنبه حضور حضرت آیتالله خامنهای فرمانده معظم قوا دانشگاه امام علی علیهالسلام برگزار مشروح خبر تصاویر تکمیلی متعاقباً منتشر',
  'politics'),
 ('یافتاد ابتدای لیگ باشگاه پرسپولیس درخواست جام هفتههای ابتدایی تیم اهدا مشخص مدیرعامل باشگاه دلیل ادعای عجیبی مطرح شب تأیید پارگی رباط صلیبی یاسین سلمانی ادعای عجیب مدعی بازیکن سه ماه آماده بازی حالی آمادگی مصدومیتی ماه',
  'sports'),
 ('لآویو قدس عسقلان اسدود شهرهای اراضی اشغالی هدف حملات موشکی قرار عملیات طوفان الاقصی شنبه گردانهای القسام رژیم صهیونیستی آغاز صهیونیست کشته تن زخمی شدهاند صدها فلسطینی حملات متجاوزانه رژیم صهیونیستی نوار غزه شهید زخمی شدهاند',
  'politics')]
```

Explanations:

As you can see, each element of the output is a tuple of processed and cleaned body of the news, and its corresponding category or label.

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

# Convert the RDD into a DataFrame
title_category_df = title_category_rdd.toDF(["title", "category"])

indexer = StringIndexer(inputCol="category", outputCol="label")

# Prepare the data
tokenizer = Tokenizer(inputCol="title", outputCol="words")
hashingTF = HashingTF(inputCol="words", outputCol="raw_features", numFeatures=10000)
data = indexer.fit(title_category_df).transform(title_category_df)
# Apply the tokenizer transformer
data = tokenizer.transform(data)
data = hashingTF.transform(data)
# TF-IDF vectorization of articles
idf = IDF(inputCol="raw_features", outputCol="features")
idf_vectorizer = idf.fit(data)
data = idf_vectorizer.transform(data)

data.show()

data = data.select('features', 'label')
```

Explanations:

Here, is a brief explanations of my code:

- **Text Vectorization using TF-IDF:**
  - Imported necessary modules from PySpark's ML library for text processing: `HashingTF`, `IDF`, `Tokenizer`, `StringIndexer`.
  - Converted the `title_category_rdd` into a DataFrame named `title_category_df` with columns `"title"` and `"category"`.
  - Applied `StringIndexer` to convert the categorical `"category"` column into numerical labels and stored the result in a new column named `"label"`.
  - Prepared the data for TF-IDF vectorization:
    * Tokenized the `"title"` column using `Tokenizer`, producing a new column named `"words"`.
    * Applied `HashingTF` to convert the word tokens into a feature vector named `"raw_features"` with a specified number of features.
    * Used the `fit` method of `StringIndexer` to transform the DataFrame.
  - Applied `IDF` (Inverse Document Frequency) to calculate the TF-IDF vectors for each article's `"raw_features"`.
  - Displayed the resulting DataFrame with columns `"title"`, `"category"`, `"label"`, `"words"`, `"raw_features"`, and `"features"`.
  - Selected only the relevant columns `"features"` and `"label"`.

title	category	label	words	raw_features	features
پایگاه اطلاع رسان...	politics	2.0	[902,1130],10000)	...,902,1130],10000)	[...پایگاه, اطلاع, ر...
رضا درویش مدیرعام...	sports	3.0	[96,275,28],10000)	...96,275,28],10000)	[رضا, درویش, مدیر...
جنبش حماس خواهان ...	politics	2.0	[77,96,110],10000)	...77,96,110],10000)	[جنبش, حماس, خواه...
بازار سرمایه فعال...	economy	0.0	[96,192,23],10000)	...96,192,23],10000)	[بازار, سرمایه, ف...
گزارش ایسنا نماین...	politics	2.0	[96,156,22],10000)	...96,156,22],10000)	[گزارش, ایسنا, نم...
بلند شماره روایتی...	culture	4.0	[46,69,96],10000)	...,46,69,96],10000)	[بلند, شماره, روا...
گزارش خبرگزاری مه...	science_and_techn...	6.0	[1,72,77,9],10000)	...1,72,77,9],10000)	[گزارش, خبرگزاری,...
رئیس دادگستری ماز...	social	1.0	[79,274,71],10000)	...79,274,71],10000)	[رئیس, دادگستری, ...
شروع عملیات طوفان...	economy	0.0	[85,95,96],10000)	...,85,95,96],10000)	[شروع, عملیات, طو...
سخنگوی قوه قضاییه...	social	1.0	[95,96,125],10000)	...95,96,125],10000)	[سخنگوی, قوه, قضا...
سردار علی پاشامحم...	economy	0.0	[167,298,4],10000)	...167,298,4],10000)	[سردار, علی, پاشا...
ایلیا حیفیا مجموع...	culture	4.0	[1,25,96,9],10000)	...1,25,96,9],10000)	[ایلیا, حیفیا, مج...
مراسم تشییع پیکر ...	culture	4.0	[96,281,46],10000)	...96,281,46],10000)	[مراسم, تشییع, پی...
رئیس مرکز تحقیقات...	social	1.0	[96,169,32],10000)	...96,169,32],10000)	[رئیس, مرکز, تحقی...
گزارش جام جم آنلا...	social	1.0	[96,119,33],10000)	...96,119,33],10000)	[گزارش, جام, جم, ...
هافبک بازیساز پرس...	sports	3.0	[15,96,103],10000)	...15,96,103],10000)	[هافبک, بازیساز, ...
نماینده دانشآموزا...	culture	4.0	[16,94,96],10000)	...,16,94,96],10000)	[نماینده, دانشآمو...
وزیر خارجه سفر لب...	politics	2.0	[96,110,34],10000)	...96,110,34],10000)	[وزیر, خارجه, سفر...
فال شمع فال فال ت...	economy	0.0	[0,1,19,43],10000)	...0,1,19,43],10000)	[فال, شمع, فال, ف...
علیرضا رضایی بازی...	sports	3.0	[1,96,281],10000)	...,1,96,281],10000)	[علیرضا, رضایی, ب...

only showing top 20 rows

Explanations:

As you can see, the features and labels(integer numbers corresponding a category) are correctly determined in the structured dataframe.

```python
(train , test) = data.randomSplit([0.75 , 0.25], seed = 202)
print("Training Dataset Count: " + str(train.count()))
print("Test Dataset Count: " + str(test.count()))
```

Explanations:

Here, is a brief explanation of what I did:

- **Train-Test Split:**
  - Split the `data` DataFrame into training (`train`) and testing (`test`) datasets.
  - Used `randomSplit` method, specifying the split ratios for training and testing datasets (75% training,

25% testing).

```
Training Dataset Count: 406923
[Stage 22:================================================>   (177 + 12) / 189]
Test Dataset Count: 135649
```

Explanations:

As it can be seen easily, we have a huge number of news to train the model and test that, which is a challenging processing and if the model wouldn't be chosen correctly, or the algorithm was inefficient, we would face memory limitations.

```
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

lr = LogisticRegression(featuresCol='features',
                        labelCol='label',
                        family="multinomial",
                        regParam=0.3,
                        elasticNetParam=0,
                        maxIter=50)

lrModel = lr.fit(train)
del train
predictions = lrModel.transform(test)

del test

predictions.select('probability','prediction', 'label').show()
```

Explanations:

Here, is a brief explanation of what I did:

- **Logistic Regression Model:**
  - Used the PySpark ML library to create a logistic regression model (`LogisticRegression`).
  - Specified features column (`featuresCol`), label column (`labelCol`), family, regularization parameter (`regParam`), elastic net mixing parameter (`elasticNetParam`), and maximum number of iterations (`maxIter`) as I saw in an article.
  - Fitted the logistic regression model on the training data (`train`) using `fit` method.
  - Performed predictions on the testing data (`test`) using the trained model.
- **Model Evaluation:**
  - Utilized the `MulticlassClassificationEvaluator` to evaluate the model's performance.
  - Displayed the model's predictions including probability, predicted label, and actual label.

```
+--------------------+----------+-----+
|         probability|prediction|label|
+--------------------+----------+-----+
|[0.43529226439635...|       0.0|  2.0|
|[0.10414441054242...|       1.0|  1.0|
|[0.21475233428591...|       1.0|  1.0|
|[0.31288095348502...|       1.0|  1.0|
|[0.27430103680319...|       1.0|  1.0|
|[0.44270762716058...|       0.0|  1.0|
|[0.43980579633832...|       1.0|  1.0|
|[0.87539365414969...|       0.0|  0.0|
|[0.47851554727533...|       0.0|  0.0|
|[0.44543264235116...|       0.0|  5.0|
|[0.19570421893707...|       1.0|  6.0|
|[2.35172757421602...|       3.0|  3.0|
|[1.76108522329332...|       2.0|  2.0|
|[0.49261500957815...|       0.0|  1.0|
|[0.16749559231725...|       2.0|  2.0|
|[0.94364771922404...|       0.0|  4.0|
|[0.22444749298890...|       1.0|  1.0|
|[0.26006443320589...|       1.0|  0.0|
|[0.62832667232219...|       0.0|  0.0|
|[0.12278664721630...|       2.0|  2.0|
+--------------------+----------+-----+
only showing top 20 rows
```

Explanations:

Obviously, the model works well, and even when it predicts something wrong, the correct prediction was in the second or third max prob. of prediction.

Finally, evaluating the model we reach a good accuracy for our specific problem with the mentioned limitations:

```
Test-set Accuracy is :  0.6380619390892938
```

Twitter Data:

Near-Duplicate Detection Streaming

Explanations:

This code which I developed, is a program that uses streaming and hashing techniques to detect spam tweets from a JSON file and simulates atreaming processes on huge data coming. The code is a novel method for spam detection in Twitter data using streaming and hashing techniques. Also, can handle large volumes of data and perform near-real-time analysis. The code uses MinHash to estimate the similarity between tweets and label them as spam or ham based on a threshold and evaluates the performance of the spam detection method by counting the number of spam and ham tweets in each batch.

The method filters out the retweeted tweets, as they are not original content, and focuses on the tweets from unverified users, as they are more likely to post spam. It uses MinHash, a type of locality-sensitive hashing (LSH), to estimate the similarity between tweets based on their Jaccard similarity. Jaccard similarity is a measure of how many words two tweets have in common, divided by how many words they have in total.

MinHash can generate hash values for tweets that reflect their Jaccard similarity, such that tweets that have more words in common are more likely to have the same or similar hash values than tweets that have fewer words in common. The method compares the hash values of the tweets with a hash table, which stores the hash values of the tweets that have been processed, and labels the tweets as spam or ham (non-spam) based on a threshold. If the hash value of a tweet already exists in the hash table, or is very close to an existing hash value, the tweet is labeled as spam, as it is a near-duplicate of a previous tweet. Otherwise, the tweet is labeled as ham, and its hash value is added to the hash table.

The method uses streaming to process the data in batches of 1 second each, and evaluates the performance of the spam detection by counting the number of spam and ham tweets in each batch. The method can detect spam tweets that have near-duplicate content, which are often unwanted, irrelevant, or malicious, and filter them out from the legitimate tweets. The method can also handle large volumes of data and perform near-real-time analysis, which are important for dealing with the dynamic and massive nature of Twitter data

```python
import time
from datasketch import MinHash, LeanMinHash
import xxhash
from pyspark.streaming import StreamingContext

hash_table = set()

def minhash(seq, num_perm=256):
    m = MinHash(num_perm=num_perm, hashfunc=xxhash.xxh64_intdigest)
    for s in list(seq):
        m.update(s.encode('utf8'))
    return LeanMinHash(m)

# Define a function to check if two tweets are near-duplicate
def is_near_duplicate(tweet1, tweet2):
  # Calculate the Jaccard similarity between the MinHash values
  jaccard_similarity =  tweet1.jaccard(tweet2)
  # Define a threshold for near-duplicate, e.g. 0.95
  threshold = 0.95
  # Check if the Jaccard similarity is greater than or equal to the threshold
  if jaccard_similarity >= threshold:
    # The tweets are near-duplicate
    return True
  else:
    # The tweets are not near-duplicate
    return False


# Create a StreamingContext
ssc = StreamingContext(sc, 1) # 1 second batch interval


processed_tweets = tweets_json.filter(lambda x: x['tweet_type'] != 'retweeted').map(lambda x: (x["text"], x.get('user').get('verifi


samples, _ = processed_tweets.randomSplit([0.0005, 1 - 0.0005], seed=42)


# Create a RDD queue from the JSON RDD
rddQueue = samples.randomSplit([0.5, 0.5], seed=42)

# Create a DStream from the RDD queue
inputStream = ssc.queueStream(rddQueue)

# Define a function to process each RDD in the DStream
```

```python
def process_rdd(rdd):
  # Loop through each tweet in the RDD
  spam_count = 0
  total_count = 0
  for tweet in rdd.collect():
    total_count += 1
    #if tweet[2] != 'retweeted':
    if tweet[1] != True:
        for previous_tweet in hash_table:
              # Check if the tweet is near-duplicate
          if is_near_duplicate(minhash(tweet[0]), previous_tweet):
      # Label the tweet as spam
            print(f'The tweet with ID {tweet[2]} and below text is spam:\n')
            print(tweet[0])
            spam_count += 1
            continue
    hash_table.add(minhash(tweet[0])) # Add verified tweets which are not retweet of another tweet

  print(spam_count)
  print(total_count)


# Apply the function to each RDD in the DStream
inputStream.foreachRDD(process_rdd)

# Start the streaming computation
ssc.start()
#time.sleep(6)
ssc.stop(stopSparkContext=True, stopGraceFully=True)
```

Explanations:

I should notice some important points again:

At the very beginning, I deleted tweets that were retweets of other tweets because they would probably have been identified as spam because they were reference tweets just like other retweets.

Also, from an article, I got the min-hashing function, which is the most optimal and memory efficient mode possible, because the hash stores tweets' texts as LeanMeanHash datasketch object, which is very optimal in terms of time and memory.

In addition, the focus of the algorithm is on tweets whose user is not verified because the probability of them being spam is much higher and rarely a verified account spams.

Now, the output for a sample of data is:

```
0
602

The tweet with ID 1735210946996363626 and below text is spam:

… به مادر قول داده بود بر می گردد
چشم مادر که به استخوان های بی جمجمه افتاد
لبخند تلخی زد و گفت :
… بچه م سرش می رفت ولی قولش نمی رفت
#وداع_با_لاله‌ها
#مهمان_مادر
The tweet with ID 1735020260640633042 and below text is spam:

ایرانی نبودن ولی دوشادوش ایرانیان مبارزه کرد و جان داد، آزادی و آزاد اندیشی مرز نمیشناسه
رضا سروری
ستاره تاجیک
هارون صدیقی
و بقیه اونایی که گمنام موندن https://t.co/MigAOoldpe
2
605
```

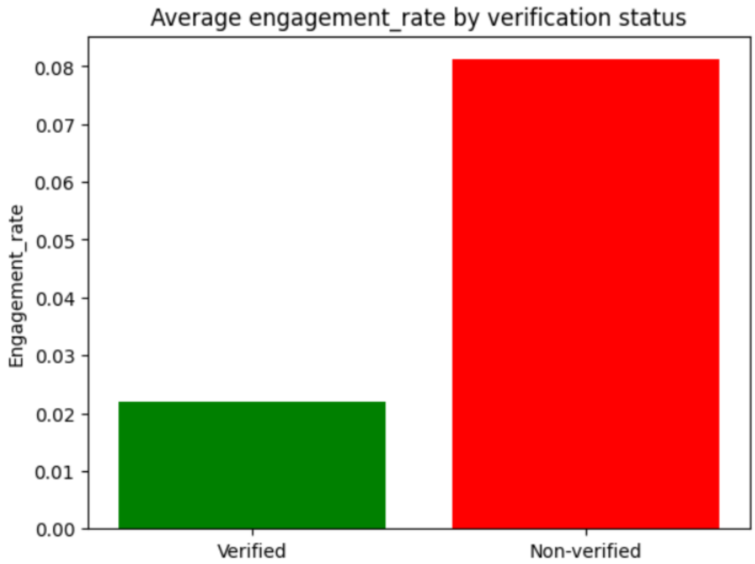
Explanations:

It is reasonably correct. Because, it detected no spam tweet from 602 elements of first streaming RDD, but detected 2 spams from 605 elements of second streaming RDD which you can see their ID and text.

More Data Analysis

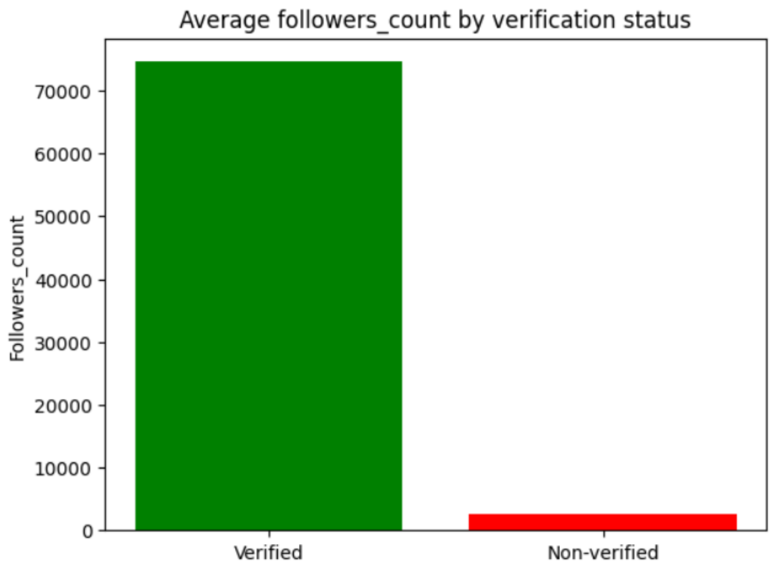
Explanations:

In this part, I analyzed the average engagement rate between two groups of users who have been verified or not. According to the output below, we come to an interesting and at first glance strange point. The average engagement rate of unverified users is about 4 times that of verified ones! While we expect users to go towards those who are more approved and interact with them. Of course, we expect it right. This was also strange for me at first, and then I did another analysis on the data and found out the reason, which I will explain in the next part.



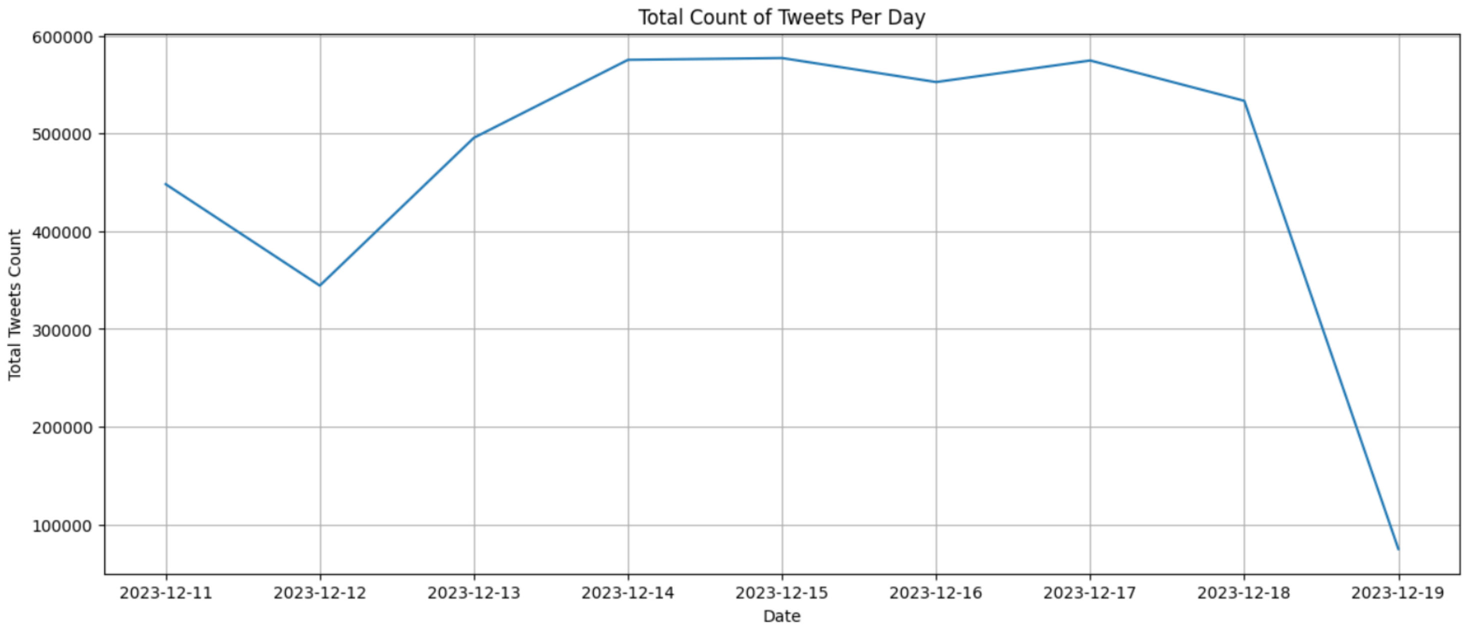
Explanations:

In the figure below, which is drawn to compare the number of followers of two groups, we see that the number of followers of verified users is much higher than the others, as we expected. But out of this number, probably many of them are not active and do not like or reply. Because the number of interactions is divided by the number of followers to get the engagement rate, the engagement rate of verified users decreases. This is more logical. Because we are looking to compare the active percentage of followers. Naturally, those who follow unverified people are very active people because not everyone does this. But, well, everyone follows verified users, and naturally the percentage of their followers' activity is lower.



Explanations:

Also, Here I brought the timeline of the tweets. However, we can't have a good inference from that because it is just for 9 days.



Clustering using Hashtags

Explanations:

In this part, to cluster the tweets according to their hashtags, I first separated the hashtags of each tweet, then using RDD, I made a matrix of hashtags and tweets, each column of which represents a tweet, and each row represents a specific hashtag. The matrix is completely composed of zeros and ones, and its element (i, j) will have a value of one if tweet j has hashtag i, and zero otherwise. Now, since we had 1.5 million unique hashtags in the data (as I have determined it in the code and you can see), I decided to do something similar to PCA and store the 200 most frequent hashtags for each tweet, which would have a vector of 200, otherwise processing would be impossible. Then, I divided them into 11 clusters using KMeans algorithm for data clustering in Euclidean space.

The reason for choosing the number 11 was that I observed that tweets are from 11 different categories, so probably 11 clusters are the most optimal mode. It was the same, and as you will see several examples of cluster 1 below, they are all similar

```
import re

def hashtag_extractor(text):
    return re.findall(r'#(\w+)', text)

hashtag_tweets = tweets_json.map(lambda x: (x['id'], hashtag_extractor(x['text']))).filter(lambda x: x[1])

hashtag_tweets.take(20)
```

Explanations:

Here, is a brief explanation of what I did:

- **Hashtag Extraction:**
  - Defined a function (`hashtag_extractor`) to extract hashtags from a given text using a regular expression.
  - Applied the `hashtag_extractor` function to the `tweets_json` RDD to create a new RDD

(hashtag_tweets).

- Filtered out tweets without hashtags.

Here is some samples of the output:

```
[('1736536753815834746', ['رضاشاه_دوم', 'جاویدشاه']),
 ('1736536756915249313',
  ['مریم', 'مریم_رجوي', 'قیام_تا_سرنگونی', 'مرگ_بر_خامنه']),
 ('1736536767254134968', ['وداع_با_لاله', 'فرزندان_زهرا']),
 ('1736536771129741402',
  ['جاوید_رضاشاه_ذوم', 'ننگ_بر_فتنه۵۷', 'ننگ_بر_سه_فاسد_ملا_چپی_مجاهد']),
 ('1736536778423607618', ['FreeHamidNouri']),
 ('1736536778704875794', ['مهسا_امینی']),
 ('1736536782022246546', ['غزة', 'فلسطین', 'انحلال_سپاه_پاسداران']),
 ('1736536778926919959', ['ساسی']),
 ('1736536785381908978', ['جاوید_رضا_شاه_دوم_پهلوی_سوم']),
 ('1736536785088561581', ['توماج_صالحی', 'FreeToomaj']),
 ('1736536787734941732', ['ساسی']),
 ('1736174366721728883', ['اسیر', 'جنگغزه']),
 ('1736174369334829269', ['فاطمة', 'شهادت', 'زن']),
 ('1736174373843689895', ['سعدی']),
 ('1736174379187175714', ['حسین_نعمتی', 'شهرک_اکباتان', 'دادبان']),
 ('1736174380877582343', ['بی']),
 ('1736174380634275950',
  ['حقوق_بشر', 'مریم_رجوی', 'آری_به_جمهوری_دمکراتیک', 'Iran']),
 ('1736174393691357422', ['مهسا_امینی']),
 ('1736536803673346164', ['KingRezaPahlavi']),
 ('1736536800284356621', ['جاوید_شاه'])]
```

```python
# Flatten the hashtag_tweets RDD to get a list of hashtags
hashtags = hashtag_tweets.flatMap(lambda x: x[1])

# Count the hashtags and their frequencies
hashtag_counts = hashtags.countByValue()

# Sort the dictionary by the values in descending order
sorted_hashtag_counts = sorted(hashtag_counts.items(), key=lambda x: x[1], reverse=True)

# Take the first 200 items
top_200_hashtags = sorted_hashtag_counts[:200]

# Create a new list of only the hashtags
unique_hashtags = [x[0] for x in top_200_hashtags]

# Modify the binary_vector function to check if the hashtag is in the unique_hashtags list
def binary_vector(hashtags):
    # Create a vector of zeros with the same length as the unique_hashtags list
    vector = [0] * len(unique_hashtags)

    # Loop over the hashtags and set the corresponding index to one if the hashtag is in the tweet and the unique_hashtags list
    for hashtag in hashtags:
        # Check if the hashtag is in the unique_hashtags list
        if hashtag in unique_hashtags:
            # Find the index of the hashtag in the unique_hashtags list
            index = unique_hashtags.index(hashtag)

            # Set the vector value at that index to one
            vector[index] = 1

    # Return the vector
    return vector

# Apply the binary_vector function to each element of the hashtag_tweets RDD
binary_tweets = hashtag_tweets.map(lambda x: (x[0], binary_vector(x[1])))

# Check the result
binary_tweets.takeSample(False, 1, 12)
```

Explanations:

Here, is a brief explanation of what I did:

- **Hashtag Frequency Analysis:**
  - Flattened the `hashtag_tweets` RDD to create a list of hashtags using the `flatMap` transformation.
  - Counted the occurrences of each hashtag using `countByValue()`.
  - Sorted the hashtag counts in descending order.
  - Selected the top 200 hashtags based on their frequencies.

– Created a list of unique hashtags from the top 200 hashtags.

- **Binary Vector Representation:**
  – Modified the `binary_vector` function to create a binary vector representation of hashtags. The vector contains ones and zeros, indicating the presence or absence of each unique hashtag.
  – Applied the `binary_vector` function to each element of the `hashtag_tweets` RDD to create a new RDD (`binary_tweets`) with tweet IDs and their corresponding binary vectors.

Here is some samples of the output:

```
[('1735167478127604210',
  [1,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
```

```
from pyspark.mllib.clustering import KMeans

# Specify the parameters for the K-means algorithm
k = 11 # The number of clusters
maxIterations = 100 # The maximum number of iterations
initializationMode = "random" # The initialization mode
seed = 42 # The seed for random initialization
epsilon = 0.01 # The convergence criterion

# Create a KMeans object with the specified parameters
model = KMeans.train(binary_tweets.map(lambda x: x[1]), k=k, maxIterations=maxIterations, initializationMode=initializationMode, se

# Train the KMeans object on the binary_tweets RDD

# Predict the cluster label for each tweet based on its binary vector
predictions = model.predict(binary_tweets.map(lambda x: x[1]))

# Zip the tweet ID, the binary vector, and the cluster label together
labeled_tweets = binary_tweets.zip(predictions).map(lambda x: (x[0][0], x[1]))
```

Explanations:

Here, is a brief explanation of what I did:

- **K-Means Clustering:**
  – Specified parameters for the K-means algorithm, including the number of clusters (`k`), the maximum number of iterations (`maxIterations`), initialization mode (`initializationMode`), seed for random initialization (`seed`), and the convergence criterion (`epsilon`).
  – Created a KMeans object (`model`) with the specified parameters.
  – Trained the KMeans model on the `binary_tweets` RDD, using the binary vectors as features.

- Predicted the cluster label for each tweet in the `binary_tweets` RDD based on its binary vector using `predict()`.
- Zipped the tweet ID, the binary vector, and the cluster label together to create a new RDD (`labeled_tweets`).

Here is some samples of the output:

```
[('1736536753815834746', 0),
 ('1736536756915249313', 0),
 ('1736536767254134968', 0),
 ('1736536771129741402', 5),
 ('1736536778423607618', 0),
 ('1736536778704875794', 0),
 ('1736536782022246546', 0),
 ('1736536778926919959', 0),
 ('1736536785381908978', 0)]
```

Explanations:

The result is reasonable. Because the cluster with tag of zero, is the cluster of political tweets, which make most of this data.

```python
# Join the labeled_tweets RDD with the tweets_json RDD
joined_tweets = labeled_tweets.join(tweets_json.map(lambda x: (x['id'], x['text'])))

# Filter the joined_tweets RDD by the cluster label 5
cluster_1_tweets = joined_tweets.filter(lambda x: x[1][0] == 1)

# Extract only the text from the cluster_5_tweets RDD
cluster_1_texts = cluster_1_tweets.map(lambda x: x[1][1])

cluster_1_texts.take(9)
```

Explanations:

Using the above code, I validated the results.

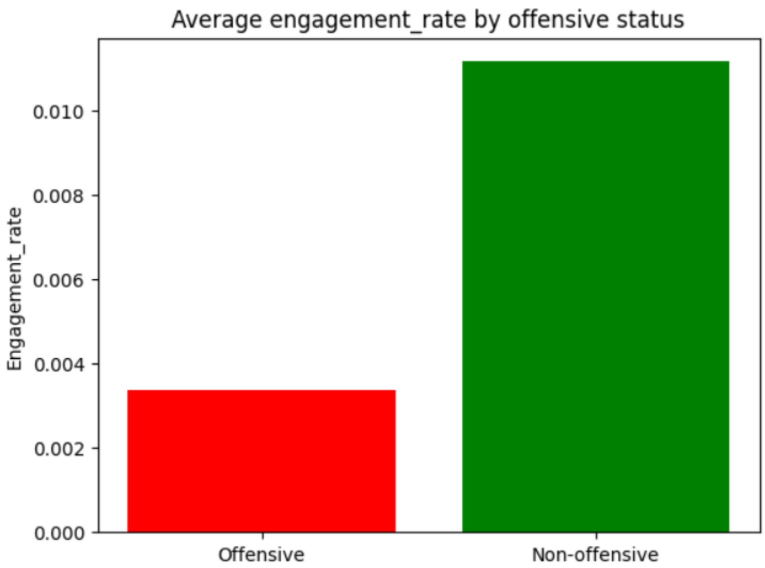
Here is some samples of the cluster with tag of 1, and clearly, all of them are about the same topic of martyrdom or Hazrate Zahra and Fatemieh:

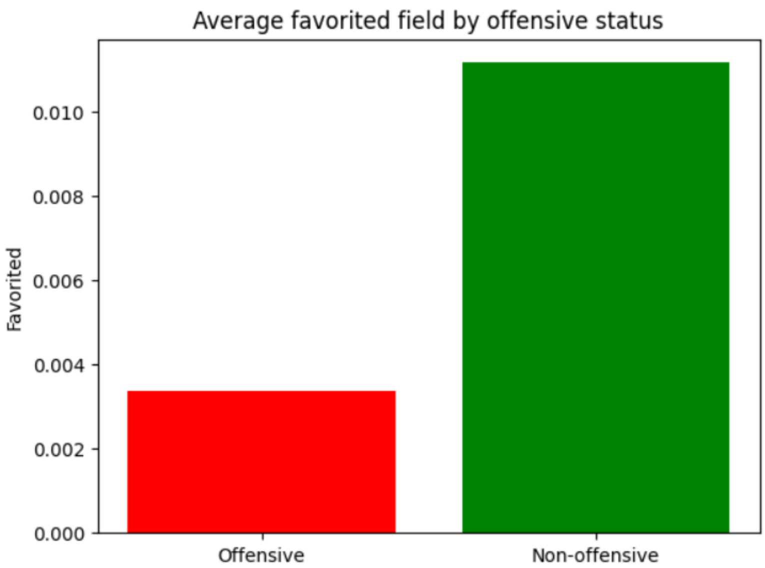
```
['/ سلامت می کنم / سلام ای مادر خوبم / سلام ای عفت و ایمان / سلامت می کنم خانوم / سلامت می کنم بانـــــو / سلامت می کنم مادر / سلامت می کنم….#فاطمیون
 'جان من و جان همه شهیدان ما، ارزش فدا شدن در راه این ملت را دارد.\n#وداع_با_لاله\u200cها\n#فاطمیون https://t.co/BqOPqSeY9c',
 'میهمان مان خواهند بود لاله هایی که برای سربلندی میهن از جان خود گذشتند\xa0انقلاب اسلامی را تضمین کردند همه مدیون آنهاییم فارغ از حزب و گروه
 'اعتش می شناسند. او گرچه معلوم نیست چطور به شهادت رسیده اما این مهم است که به این مقام نائل شده و مردم ما خوب بلدند قدر این مقام را بدانند
 'سلام بر آن\u200cهایی که\nرفتند تا بمانند و نماندند تا بمیرند\nو تا ابد به آنانکه پلاکشان را از گردن خویش درآوردند\nو بی مزار بمانند مدیونیم
 'فقط دم\u200cزدن از شهدا افتخار نیست! باید زندگیمان، حرفمان، نگاهمان ، رفاقتمان، بوی شهدا بدهد.\n\n#وداع_با_لاله\u200cها\n#فاطمیون https:/
 'هر چه میکشیم از دست این نام هاست، و این را شهید گمنام خوب فهمیده است\n#وداع_با_لاله\u200cها \n#فاطمیون',
 '"سالروز شهادت شهید مدافع حرم #فاطمیون "خداداد رضایی\n\nولادت: ۱۳۵۷\nشهادت: ۱۳۹۵/۹/۲۷\nمحل شهادت: تدمر\nمزار: تهران/بهشت زهرا(س)\n#صلوات #شهدا
 'وداع مادر با لاله ها همزمان با سالروز وداع حسنین(ع) با حضرت مادر(س)....\n#وداع_با_لاله\u200cها\n#فاطمیون https://t.co/r471fVFgLv']
```

NLP field Analysis

Explanations:

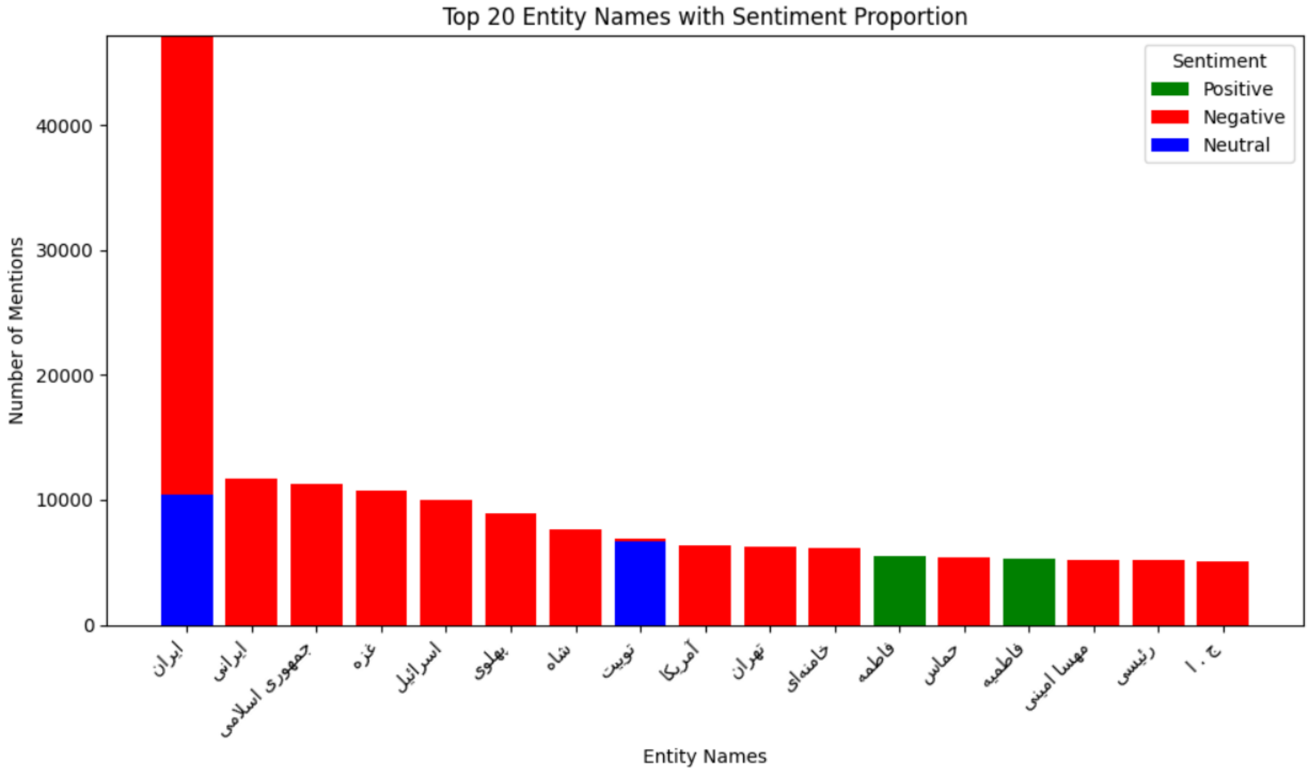
In this part, I examined the data that had the nlp field so that we can extract more useful and deeper information from the tweet data. In the first graph, the average engagement rate of offensive tweets has been compared with other tweets. Naturally, followers liked non-offensive tweets more. In the second graph, the same thing is checked for interest in tweets, which results are similar and show that users like non-offensive tweets more.

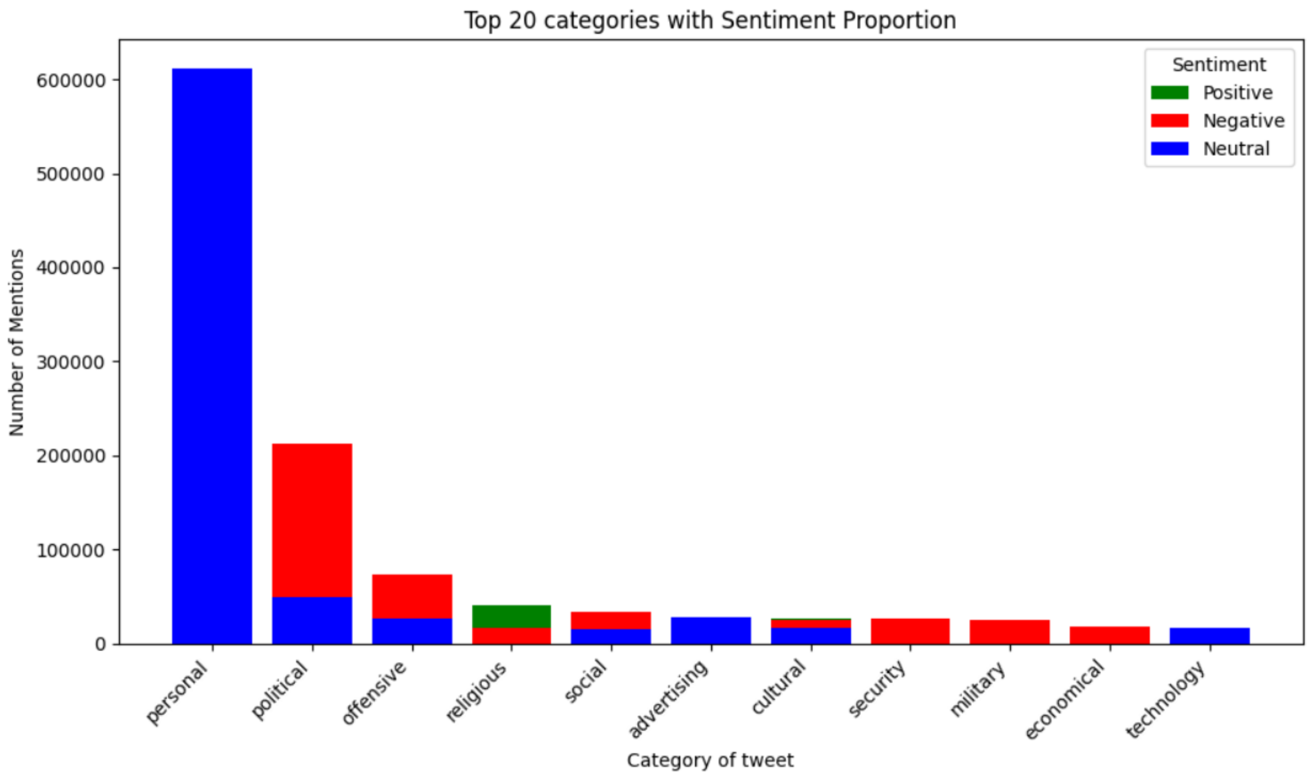




Explanations:

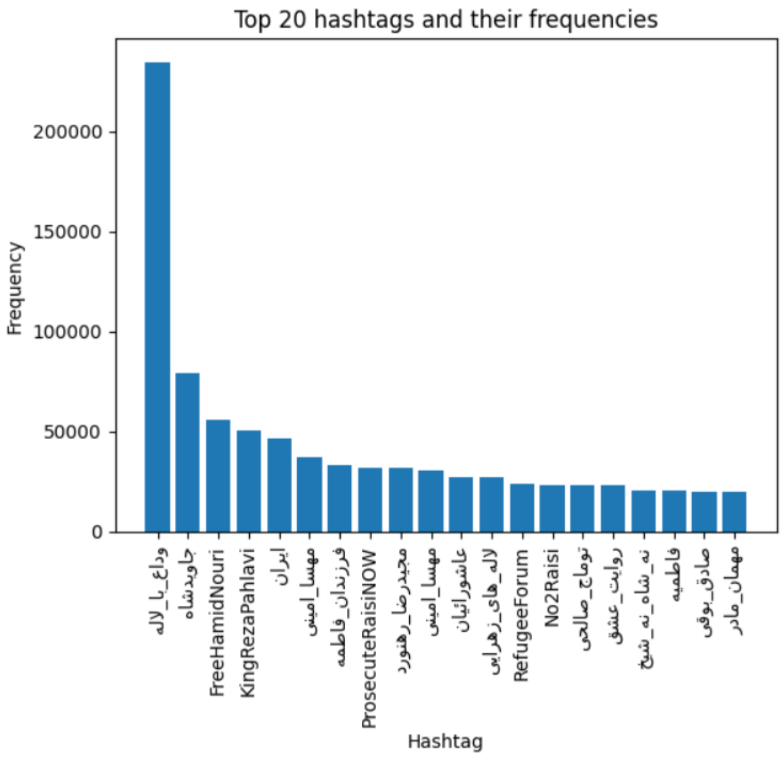
In the next two graphs, for 20 most frequent entity names, and then for 20 most frequent categories, the percentage of different sentiments in their tweets has been checked. The first graph shows that most of the tweets related to political entity names have a negative sentiment and are critical or protest. But religious people, for example Hazrat Fatemeh, have a positive sentiment and there is nothing like a tweet. The same is true for categories, and religious are the categories that have a significant share of positive tweets.





Explanations:

Also, I brought the most frequent hashtags in a plot to better visualize, which as you can see, the political hashtags and related hashtags are the most frequent. This indicates the data to be user-tracked as we know.


Top 20 hashtags and their frequencies
Frequency
0
50000
100000
150000
200000
Hashtag
وداع_با_لاله
جاویدشاه
FreeHamidNouri
KingRezaPahlavi
ایران
مهسا_امینی
فرزندان_فاطمه
ProsecuteRaisiNOW
مجیدرضا_رهنورد
مهسا_امینی
عاشورائیان
لاله_های_زهرایی
RefugeeForum
No2Raisi
توماج_صالحی
روایت_عشق
نه_شاه_نه_شیخ
فاطمیه
صادق_بوقی
مهمان_مادر