Sentiment Analysis of The Ukraine–Russia War

Using Hadoop, Hive, and Spark to do sentiment analysis on tweets related to the Ukraine-Russia war.

Ola Andrè Flotve
University of Stavanger, Norway
oa.flotve@stud.uis.no

Håvard Moe Jacobsen
University of Stavanger, Norway
ham.jacobsen@stud.uis.no

ABSTRACT

The ever increasing amount of data leads to large amount of information being available. A popular way to utilize this information is through the use of sentiment analysis, but with a lot of data this becomes expensive. We want to solve this problem using Hadoop with Spark and Hive. In this paper we explored this method with Twitter data which we gathered from the Ukraine-Russia war.

KEYWORDS

Sentiment Analysis, Hadoop, Spark, Hive, Ukraine, Russia

1 INTRODUCTION

As more and more data becomes available the need to extract essential information from large amounts of data becomes a critical task for anyone who wants to perform analysis. The size of computational power needed means that using a standalone computer is no longer feasible.

This is where we look to Apache Hadoop [14], Spark [32], and Hive [16]. With the use of Apache Hadoop one can combine the computational power of multiple computers into one cluster, and by integrating Apache Spark one top of the Hadoop framework we get enhanced clusters which are faster and provide more functionality [9]. When working with huge quantity of data the need for efficient storage and easy access becomes important, and this is where we introduce Apache Hive. Hive is a data warehouse software which offers the ability to both read and write large number of data which is distributed into multiple machines [30]. The software also supports a SQL-like language which offers easy access to the datasets. The combination of these tools make for a great platform to work with big data.

A popular way of extracting information from data is through sentiment analysis. The idea behind is to examine text and determine if the sentiment is positive or negative. Sentiment analysis can also go by the name opinion mining which is fitting since one of its main applications is to obtain the opinion of customers [21].

The most essential work of the paper is the following:

- Gather twitter data from the Ukraine–Russia war through the Twitter API [40] with the help of Tweepy [37].
- Store the raw data with Apache Hive in HDFS.
- Preprocess the data through Apache Spark.
- Perform sentiment analysis on preprocessed data.
- We then want to evaluate the following:
  - Is it possible to detect major events in the war by looking at time-series plots?
  - Is it possible to detect trends before major events occur?
  - Is it possible to detect a change in sentiment from before the war until now (April 2022)?

Supervised by Tomasz Wiktorski and Jayachander Surbiryala.

Project in Computer Science (DAT500), IDE, UiS
2018.

2 BACKGROUND

In this section we will introduce the frameworks and methods we used in the project.

2.1 Twitter

Twitter is a social media platform which was released to the public in 2006 and has grown to be one of the most used social media platforms [35]. The platform lets users share thoughts and opinions in form of text, smileys, media and URLs. Each of these composed texts is called a tweet. Each tweet is limited to 280 characters which makes tweets short and concise in manner [41]. By the use of hashtags a user can indicate that their tweet relates to a specific topic or category [15].

2.1.1 Twitter API. Twitter offers an API which lets you search tweets of interest by the use of http requests containing a query [40]. The query can contain a variety of attributes such as user id, tweet id, keywords, hashtags and more. When performing a query you will receive a 'page' which contains a maximum number tweets together with a *pagnation_token* which lets you access the next page of results. Unless you have the 'Academic Research access' you will not be able to view tweets older than a week and the max size of pages received from the http request will be 100 instead of 500.

2.2 Tweepy

Tweepy is an open-source python library which lets you access the Twitter API via python code [37]. It is easy to use and reduces the overhead of handling pages received from requests to one of Twitters API endpoints. Having a python interface also gives us the possibility to use other python libraries to do minor tweaks to the data before moving it through the pipeline.

```
client = tweepy.Client(bearer_token=twitter_api_token,
    wait_on_rate_limit=True)
for response in tweepy.Paginator(client.search_all_tweets
    , query, ...):
    for tweet in response.data:
        # Handle tweet
```

Listing 1: Tweepy example usage

2.3 Apache Hadoop

Apache Hadoop is a framework which makes it possible to combine multiple machines into a cluster specially designed for big data management [14]. Instead of upgrading a single machine with

expensive hardware components, Hadoop offers you the possibility of horizontal scaling. The Hadoop framework is divided into some main components:

2.3.1 Hadoop Distributed File System. The Hadoop Distributed File System (**HDFS**) splits data into 'blocks' having a default maximum size of 128MB. These blocks are stored in a distributed manner across the nodes in the cluster. To prevent data loss in the case of faulty machines, the blocks are stored with a default replication factor of three. Meaning that every single block is stored on three different nodes.

2.3.2 MapReduce. MapReduce is a data-processing framework which is optimized for processing big datasets in parallel on a distributed cluster [8]. By the use of a mapper function data is split into smaller parts containing a (key, value) pair e.g. (word_a, 1). The (key, value) pairs are subsequently sent through a reduce function which combines the pairs into meaningful results e.g. a count of how many times words have occurred in a text.

There are usually many jobs running simultaneously which all requires resources found on the cluster nodes. To efficiently manage those resources, we have the third component of hadoop, YARN.

2.3.3 YARN. Yet Another Resource Negotiator (**YARN**) is responsible for providing computational resources for application executions.
There are five steps to run an application on YARN [31]:

(1) An application is submitted to the Resource Manager
(2) The Resource Manager allocates a container for the application, the belonging Application Master is started and is responsible for the entire life cycle of the particular application
(3) The Application Master contacts the related Node Manager to use the given container to launch a application related task
(4) The Node Manager launches the container and monitors it's resource usage and progress
(5) The container executes the Application Master

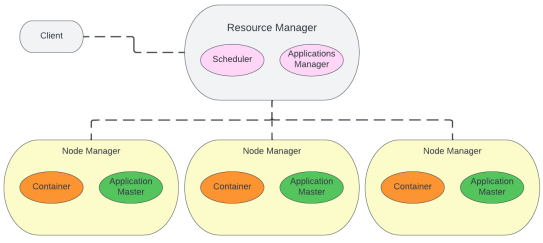

Figure 2.1: YARN architecture illustrated

YARN's main components:

- There are usually a single **Resource Manager** in a cluster. It knows the location of the data nodes and how many resources they have. The Resource Manager runs multiple services, the most important being the **Scheduler**, which decides how to distribute the cluster's resources in a way which maximizes the cluster's utilization. Another important service located in the Resource Manager, is the **Applications Manager**. It's job is to accept job submissions, negotiate containers for executing jobs, and restarting the Application Master Container on failure.
- The **Application Master** is a framework specific process and could be could be written in any language. It negotiates resources with the Resource Manager and works with the Containers provided by the Node Managers.
- There are usually multiple nodes in a cluster which all have their own **Node Managers**. Node Managers handles the node and manages the resource usage inside a specific node. At runtime, the Node Manager takes instructions from the Resource Manager. When initiated, it sends an initialization-signal to the Resource Manager informing it of the node's existence. Thereafter a routinely heartbeat signal.
- Every node in the cluster usually consists of several **Containers** which each holds part of the node's physical resources (i.e. RAM, CPU and bandwidth). Containers are used by the client to run a program. The application manager presents the Container to the Node Manager on the node where the Container has been allocated. Thereby granting it access to the resources.

[26][22][31]

An instance of a Hadoop YARN cluster can be used to run multiple types of applications, some of them are MapReduce, Hive and Spark.

2.4 Apache Hive

As mentioned earlier Apache Hive is a data-warehouse software which works in cooperation with Hadoop and can also be used with Spark. Its architecture can be divided into four main components:

- The first component is the **Hadoop core components** which is made up of the HDFS and MapReduce. The HDFS is used to store the data which are contained in the tables, and to access this data one can execute queries in Hive which will then be converted into MapReduce jobs [29].
- The next component is the **Metastore**. The Metastore is used as a namespace for the tables. Its the location where Hive stores its metadata such as the information about the rows and columns in the tables [29]. The Metastore is by default stored with Derby, but these days its more common to use MySQL [12].
- The third component is the **Drivers** which is where the queries arrive after they are executed by the users. It consists of a Hive Compiler, Optimizer, and Execution Engine which is responsible for parsing the query, creating a execution plan, optimize it, and in the end execute the plan [10].
- The last component is called the **Hive clients**, and is where the queries from the users are submitted through an interface such as the Hive CLI [29].

Hive's architecture lays the foundation for a great software which makes for easy handling of huge amounts of data. The way it cooperates with Hadoop through the cluster and HDFS results in a fast and optimized way for retrieval and storage of data. Hive can

easily be integrated into Spark which makes reading and writing in between data processing a simple task, and also makes it possible to utilize the benefits of Spark when doing transformations on data.

2.5 Apache Spark

Spark is an multi-purpose engine used for machine learning and big data analysis from a single-node computer, or a cluster of nodes [32]. Spark applications requires a cluster manager and distributed storage system like YARN and HDFS and can be written in a variety of languages [34][32].

The main components of Spark applications are RDDs which stands for Resilient Distributed Datasets [45]. An RDD is an abstraction for a read-only collection of data distributed over a cluster. In order to create an RDD-object, one needs to specify where in the distributed file system the data is stored. When using HDFS as the distributed file system, the data can be stored in files or in Hive a database. The target data can then be read by using a path to the file(s) or by using a Hive-query to select the relevant data.

Transformations like *map*, *reduce*, *foreach* performed on the RDD are done in a lazy manner, meaning that they are not actually performed until one tries to read, save, or forcefully make the transformations by calling a collect() or count() on all the objects of the RDD [45]. One may therefore perform multiple transformations on the RDD object before the execution process is started.

Spark addresses some of the weaknesses of a single MapReduce application running on Hadoop which are **Iterative jobs** and **Interactive analysis** [45]. Iterative jobs on a bare-bone MapReduce implementation could be slow as each MR job has to both read and write the data before the next iteration starts. Instead of having to run a separate MR job to load and store data in between analysis, a user would be able to load a dataset into memory across a number of machines and query it repeatedly which also removes the unnecessary overhead for interactive analysis [45].

2.6 Sentiment Analysis

Earlier in the paper there was given a brief but concise introduction into what sentiment analysis is. This section will give the reader a better understanding of the technical tools used to perform the analysis.

2.6.1 TextBlob. As stated in its documentation; TextBlob is python library for processing of textual data [36]. It provides a simple API for performing different types of natural language processing such as sentiment analysis. TextBlob's sentiment analysis is performed by using a training set from the Natural Language Toolkit [28][24]. Text is then provided and sentiment score will be calculated for each word, and the final result will be a weighted average over all the words. The output from TextBlob will then be a tuple consisting of a polarity score between -1 and 1, and a subjectivity score between 0 and 1 which gives and indication on how personalized the opinion is.

2.6.2 VaderSentiment. VaderSentiment which stands for Valence Aware and sEntiment Reasoner is a sentiment analysis tool which uses a lexicon combined with rules to produce its results [17]. VaderSentiment is also specially created to perform well with data from social media [17]. The most used output from the analysis is the compound score which is calculated by summing valence scores from each word. The sum is then normalized so that one gets output between -1 and 1.

2.6.3 WordList. Using wordlists for sentiment analysis is a simple yet effective method if you are dealing with large datasets. For our project, we used the AFINN wordlist which contains 2477 words where each word has a sentiment rating of -5 to +5 [23] [6]. Neutral words are not present in the list.

2.7 Related Work

This section will give a brief introduction to some of the previous works which inspired us to do this project.

Gosain [13] created a project for Twitter Sentiment analysis using Hadoop. With the integration of Hive and Flume he fetches twitter data and stores it. The sentiment analysis is then performed by using a dictionary. The analysis focuses on which people that are the most influential as well as what the opinion of people from different countries are about a topic.

Ripamonti [25] used a machine learning model called Word2vec which was trained on a Twitter dataset and then later used to classify tweets into negative, neutral, and positive categories. The analysis was performed with the use of a Kaggle notebook without the use of a Hadoop cluster.

3 PROPOSED METHOD

In this paper we propose a Hadoop based method of sentiment analysis optimized with Apache Spark and Hive. The data used will be collected by the use of the Twitter API.

3.1 Acquisition of Data

Twitter offers extended access to its API for academic research. To obtain this access one needs put in an application, which we did and it got accepted. With academic access one can fetch up to 10 000 000 tweets a month from the Twitter API v2 [38]. It is also possible to retrieve historical data all the way back to March 2006 [39].

The Twitter API will be used to gather data from January 2012 until April 2022. The API will filter on the keywords Ukraine and Russia which will give us the majority of the tweets related to the war.

3.2 Transferring and Organizing data

The next step will be to transfer the dataset into the HDFS which is carried out by built in commands in Hadoop. Then with the help of Spark the raw data can then be written into a Hive table.

3.3 Preprocessing of Twitter data

By the use of Spark SQL we read the data from the raw data table into a Spark dataframe. The cleaning of the data then proceeds by transforming the tweets to the desired format which is achieved in several steps, one of them being removal of special characters. After the data is processed it will then be saved into a new Hive table.

3.4 Sentiment Analysis

The preprocessed data is retrieved by Spark from the Hive table. Then by the use of different sentiment analysis library's such as TextBlob, VaderSentiment and a wordlist [23], sentiment results are produced.

3.5 Presentation of Sentiment Results

In the end the results will be presented with the use of graphs from matplotlib [20] and Seaborn [27]. Visualizations will be created with a specified time interval which will dictate which values that will be included when calculating averages. Furthermore plots for month by month and day by day will be created to see if its possible to detect major events in the conflict as well as trends.

There will also be created a average score from the three sentiment analysers which are mentioned earlier in the paper. In order to do this we need to get all three of the sentiment scores on the same scale. By default both TextBlob and VaderSentiment produces scores between -1 and 1. Due to the nature of wordlists, it is hard to assume the score range and transform this to -1, 1 scores. We will therefore be using the StandardScaler [5] from pyspark for this purpose.

4 SETTING UP THE ENVIRONMENT

This chapter will describe the set up for Apache Spark and Hive.

4.1 Hive Implementation

There are several steps in the implementation of Hive [18]. The steps used in this paper will be reviewed below:

(1) In the first step we searched for a appropriate version of Hive which would be compatible with our Hadoop version and then download it.
Note: In this project we decided to use Hive 3.1.2.
(2) In the second step we configured the Hive environment variables in the bashrc file. This included setting up HIVE_HOME and Hives bin path to make the Hive applications runnable from any directory.
(3) For Hive to work together with Hadoop Hive needs to be able to locate the HDFS. This was solved by adding the HADOOP_HOME path to the hive-config.sh file.
(4) In the next step we created two new directories in the HDFS.
  (a) The first directory is called tmp and is used by hive to store the intermediate results.
  (b) The second directory is called warehouse. This is where hive stores its tables. The new warehouse is usually situated inside a hive directory which again is in the user directory.
Note: The newly created directories need write and execute access.
(5) Step five consisted of initiating the database. The default database used in Hive is Derby which we first used, but after doing some reading we realised that derby only allows for one connection at a time, so we switched to a MySQL database. The database was created by using Hive's schematool with the keywords MySQL, and initSchema.
(6) In the last step we encountered a problem with different versions of guava files in Hive and Hadoop which resulted in Hive not working. This was easily fixed by changing the guava files to the same version.

After the completion of these steps, the Hive environment was up and running.

4.2 Spark Implementation

Our cluster is running Hadoop version 3.2.1, so we had to find a compatible version of Spark which were version 3.1.3 [33]. As with Hive, we created the shortcut $SPARK_HOME for our home path.

Once we got Spark working on the cluster, we also set up a tunnel on port 8080 which made us able to track spark jobs from our local computer which is used to monitor and control the cluster.

```
ssh -D 8080 -C -N remote.ip.add.ress
```

Subsequent to running this command on our local machine, we were able to view the Spark UI which gave us information about which applications were running on the cluster as well as information about the cluster in general. After setting up a proxy on a web browser like firefox, the UI is accessible through the browser by going to the <name of masternode>:8080.

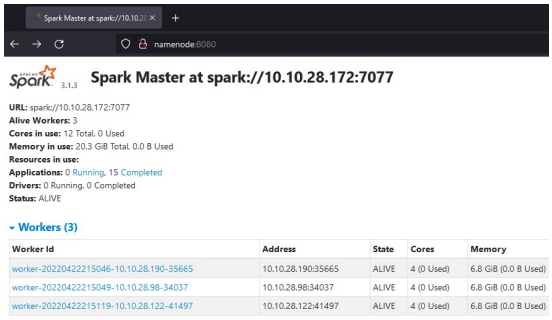

Figure 4.1: Screenshot of the Spark UI

When we got the Spark framework up and running, we wanted to create an easy ways of running applications on it. We first read about the standard pyspark shell, and the **spark-submit <pyfile>** command but wanted instead to implement a pipeline which would let us test code more interactively. To achieve this, we downloaded pyspark, then imported and ran these commands at the beginning of our python notebook:

```
spark = SparkSession \
    .builder \
    .master('spark://10.10.28.172:7077') \
    .appName('visualizing') \
    .getOrCreate()
```

Listing 2: Spark setup for python applications and notebooks

Note: 10.10.28.172 is the local ip address of our namenode.

After the initial setup we can execute jobs directly on the cluster by first querying data from the HDFS, thereafter executing jobs on this data.

4.3 Integrating Hive with Spark

For Spark to be able to communicate with Hive we had to add some properties to the hive-site.xml file which was required to be present

in both Spark and Hive. Among other things it needed a username and password from the database to be able to connect to it, and also some information on what type of database it connected to.

After we finished with this step, we added another line to the initialisation of our Spark applications as well as our notebooks: **.enableHiveSupport()** This command lets spark.sql access the Hive DB:

```
spark_df = spark.sql('select * from twitter_data.raw_data;')
```

Listing 3: Example of how we access Hive through Spark

5 IMPLEMENTATION

This section describes and illustrates the main parts of our pipeline and code.

5.1 Acquisition of Data

As mentioned in previous sections; we accessed the Twitter API via the use of the python library Tweepy. We made a function called fetch_twitter_data which takes a query, from and to-time as input and collects tweets in batches of 500 at a time (500 is the max_page_size mentioned in the Twitter API subsection). The Tweepy library gives us an abstract object we can loop over in order to access page after page.

```
def fetch_twitter_data(client, query, start_time, end_time):
    tweets_in_file = 0
    total_fetched_tweets = 0

    start_dt = datetime.strptime(start_time,'%Y-%m-%dT%H:%M:%SZ').replace(tzinfo=timezone.utc)
    end_dt = datetime.strptime(end_time,'%Y-%m-%dT%H:%M:%SZ').replace(tzinfo=timezone.utc)

    total_days = (end_dt - start_dt).days

    filename = ''  # gets set iteratively for each new file
    fields = ['id','text','author_id','created_at','geo']

    for response in tweepy.Paginator(client.search_all_tweets,
                query = query,
                tweet_fields = fields,
                start_time = start_time,
                end_time = end_time,
                max_results=MAX_TWEETS_PER_PAGE
    ):
        max_date = max([tweet.created_at for tweet in response.data])
        days_fetched = (end_dt - max_date).days

        if tweets_in_file > MAX_TWEETS_PER_FILE or tweets_in_file == 0:
            tweets_in_file = 0

            filename = f'{max_date.strftime("%Y%m%d%H%M%S")}'

            write_csv(fields, filename, new=True)  # write headers to file

        tweet_list = [[tweet.id, tweet.text.replace('\n', ' '), tweet.author_id, tweet.created_at, tweet.geo] for tweet in response.data]
        write_csv(tweet_list, filename)
        time.sleep(1)
        tweets_in_file += response.meta['result_count']
        total_fetched_tweets += response.meta['result_count']

        if total_fetched_tweets > MAX_TWEETS:
            print('Max_tweets exceeded, exiting')
            break
```

Listing 4: Function used to fetch tweets

We chose to fetch the tweet_id, text of the tweet, author_id, time of creation and the geo location of the tweet. Yet we did not end up using the author_id or geo-location in this project. We loop over the abstract Tweepy object which gives us a page of tweets at a time, and write them to a csv file. When the file becomes large enough (we used MAX_TWEETS_PER_FILE = 100_000) we started writing to a new csv file. The filename for each file is the date of the first tweet in the file.

5.2 Transferring and Organizing Data

We moved the csv files to HDFS with the command: **hadoop fs -put /data_path/*.csv /data**. Thereafter we read the data from HDFS into a Spark application and saved it as a Hive table.

```
raw_data = spark.read.csv('hdfs://namenode:9000/data/*.csv', header=True)
# Cast to correct columntypes
data = raw_data.\
withColumn('id', col('id').cast(LongType())).\
withColumn('author_id', col('author_id').cast(LongType())).\
withColumn('created_at', col('created_at').cast(TimestampType()))
# Write to hive table
data.write.format('hive').mode("overwrite").saveAsTable("raw_data")
```

Listing 5: Read data from csv to Hive DB

5.3 Preprocessing

Before performing sentiment analysis on the tweets, we wanted to do multiple steps of preprocessing:

(1) Remove rows with NULL values
(2) Remove tweets which where stored on the wrong format which led to csvfields jumping one step too far. (around 70 occurrences out of 13_000_000)
(3) Translate emojis into words
(4) Filter text with the use of **pyspark.sql.functions.regex_replace()**
- Remove URLs
- Remove user mentions (i.e. @username)
- Remove all characters which are not a letter or space
- Replace multiple spaces after one another into a single space
- Remove potential spaces in the beginning of the text

```
query = '''
    SELECT *
    FROM raw_data
```

```
    WHERE text IS NOT NULL
    AND created_at IS NOT NULL
    AND id IS NOT NULL
    AND created_at >= date"2012-01-01"
    AND created_at <= date"2022-04-01"
'''
raw_data = spark.sql(query)
```

Listing 6: Filtering NULL values and wrong dates

```
data = raw_data.\
    withColumn('text', udf_translate_emojies(col('text'))
    ). \
    withColumn('text', F.regexp_replace(col('text'),
    url_regex, '')). \
    withColumn('text', F.regexp_replace(col('text'),
    user_regex, '')). \
    withColumn('text', F.regexp_replace(col('text'),
    neg_alphanumeric_regex, '')). \
    withColumn('text', F.regexp_replace(col('text'),
    double_spaces_regex, ' ')). \
    withColumn('text', F.regexp_replace(col('text'),
    start_spaces_regex, '')). \
    withColumn('text', F.lower(col('text'))
)

# write to hive table - twitter_data.proccesed_data
spark.sql('drop table if exists processed_data')
data.write.format('hive').mode("overwrite").saveAsTable("
    processed_data")
```

Listing 7: Translating emojis and Filtering with regex

For emoji processing, we used a python library called emoji which translates the unicode the emoji is written in, into what the emoji is supposed to represent (alias) [19]. As an example: an emoji symbolizing a red heart will be translated into ":red_heart:". We later noticed VaderSentiment does its own translation of emojis, nevertheless we thought it would be a good idea to do this translation for the whole data as other sentiment analysis could benefit from this [17].

5.4 Sentiment Analysis

For the analysis, we created user defined functions (UDFs) which we ran by using the **.withColumn('new_col', function('col'))** SparkDF - function. Here is an example of how we created an UDF for the TextBlob analysis:

```
sentiment_tblob = lambda text: TextBlob(text).sentiment.
    polarity
udf_tblob_sentiment = udf(lambda text: sentiment_tblob(
    text), FloatType())
sentiment_score = df.withColumn('sentiment',
    udf_tblob_sentiment('text'))

sentiment_score.write.mode("overwrite").saveAsTable("
    textblob_results")
```

Listing 8: Sentiment analysis using TextBlob as an UDF on a Spark Data Frame

We save the results from all three methods into three different tables: 'textblob_results', 'vader_results' and 'wordlist2477_results'. Thereafter we combined the different result tables into a single table called results. We also included the created_at column from the processed_data table in order to make it easier to plot the results at a later time.

5.5 Visualisation of Data

We used a python notebook connected to Spark for the visualization of plots. This gave us the possibility of testing different plotting-parameters without having to re-run scripts entirely.

We loaded the 'results' table into an Spark DataFrame. Then created two instances of this DataFrame, and scaled one of them using the StandardScaler from pyspark. In order to scale the different results columns we combined them using Spark's VectorAssembler which combines the results columns to a matrix which we use as input to the StandardScaler:

```
query = '''
    SELECT created_at, vader_sentiment,
    textblob_sentiment, wordlist2477_sentiment
    FROM results
    '''
df = spark.sql(query)

va = VectorAssembler(inputCols=sentiment_columns,
    outputCol='sentiment_combined')
temp_df = va.transform(df)
ss = StandardScaler(inputCol='sentiment_combined',
    outputCol='scaled_sentiment')
scaled_df = ss.fit(temp_df) \
    .transform(temp_df) \
    .select(['created_at','scaled_sentiment']) \
    .rdd.map(extract).toDF(col_names)
```

Listing 9: Load and Scale Results

After loading and scaling, we grouped the sentiment scores into intervals of 24 hours over the period 2012-01-01 til 2022-04-01 and take the mean of each 24-hour period.

```
scaled_df = scaled_df. \
    select(col_names). \
    groupBy(F.window('created_at', INTERVAL)). \
    mean()
```

Listing 10: Grouping Spark DF

As mentioned earlier; transformations on Spark RDDs are not performed until you read from the RDD. Because of this, and to prevent having to do the scaling transformation for each plot we were to make, we used the Spark DataFrame function .persist(). As input for this function, we used StorageLevel.MEMORY_ONLY which tells Spark to store the RDDs information fresh in memory to prevent it from having to calculate it all over again.

```
scaled_df.persist(StorageLevel.MEMORY_ONLY)
scaled_df.collect()
```

Listing 11: Usage of .persist() and StorageLevel

We also did this for the non-scaled DataFrame, to prevent having to do the grouping multiple times over. A collection of the code we used is in our github repository [11].

5.6 Finished Pipeline

Figure A.3 illustrates the pipeline from where the data fetching begins, until the end where we present our results and visualisations.

6 RUNTIME AND OPTIMALIZATION

In this section we are going to discuss the performance of the different applications we ran on our cluster.

6.1 MR with RDD vs withCoulmn in Spark DF

After we set up our cluster with all our frameworks up and running, we wanted to see what was the most optimal way of performing sentiment analysis for our dataset. We decided to do a simple benchmark before choosing the best path forward. After some research, we decided to compare two of the most commonly used methods for the usage of mapper-functions in Spark. Namely the **withColumn** dataframe function and a MapReduce implementation used on an RDD.

In order to do a pure MR job, we wanted to explode tweets having (date, text) key-value pairs into multiple (date, word) pairs, and use the word in the sentiment dictionary: (date, dictionary(word)) and finally date to get columns ['date', 'sum(word_sentiment)]. This closely resembles what we want to from the start to the end of our pipeline, so we decided to use this as our benchmark.

```
df = spark.sql('select to_date(created_at) as date, text
    from processed_data')

df.withColumn('sentiment', udf_wl_analyze('text')) \
    .groupBy(F.window('date', '1 day')).sum() \
    .select([
        F.to_date(F.col('window.start')).alias('date'),
        F.col('sum(sentiment)').alias('total_sentiment')
    ]) \
    .write.mode('overwrite') \
    .saveAsTable('daily_sentiment_DF')
```

Listing 12: Benchmark: withColumn and groupBy on Spark DF

```
df = spark.sql('select to_date(created_at) as date, split
    (text, " ") as word from processed_data')

rdd = df.rdd
rdd.flatMapValues(lambda x: x) \
    .mapValues(lambda x: wordlist[x]) \
    .reduceByKey(add) \
    .toDF(['date', 'total_sentiment']) \
    .write.mode("overwrite") \
    .saveAsTable('daily_sentiment_MR')
```

Listing 13: Benchmark: MR on Spark RDD

The runtimes were 1 minute and 2.3 minutes in favour of the dataframe implementation (A.1). Due to our satisfaction with the speed of the Spark dataframe, we did not see any use in spending a lot of time implementing it in a different way.

6.2 Sentiment Analyzers

We implemented a python file with functions for TextBlob and VaderSentiment analysis, and a class for wordlist analysis. The TextBlob fucntion looked like this:

```
def sentiment_tblob(text):
    return TextBlob(text).sentiment.polarity
```

Listing 14: TextBlob analyser

This function worked fine, so we did a similar approach for the Vader implementation:

```
def old_sentiment_vader(text):
    va = SentimentIntensityAnalyzer()
    return va.polarity_scores(text)['compound']
```

Listing 15: Naive Vader implementation

This is when we encountered a problem: while the TextBlob analyser used around 5 minutes to analyse the whole dataset, the Vader implementation used 3/4hours[1]. We soon figured that the initialization of the Vader-class was unneeded in the UDF, and was the reason for the overlong runtime. We then decided to create the class object outside the UDF instead, and pass the class-object's function to the UDF.

```
analyzer = SentimentIntensityAnalyzer()
va = lambda text: analyzer.polarity_scores(text)['
    compound']
udf_va = udf(lambda text: va(text), FloatType())
```

Listing 16: Proper Vader implementation

This change of implementation drastically changed the runtime to around 4.5 minutes[2].

6.3 Cleaning the Data

After retrieving the data from Hive, we used the **withColumn** mapper function for the remaining emoji and regex cleaning (please view the preprocessing.py file on our github for details[11]). This process took around 4 minutes which can be seen in the A.2 image.

6.4 Visualisation

We yet again queried data from a Hive table, this time from the 'results table' we firstly scaled the data, then grouped it into 24 hour intervals and stored this in a Spark dataframe for plotting. We passed this dataframe to a plotting function (one time for each plot) we made, which then forced the transformations in the dataframe to be executed by calling the toPandas() Spark dataframe function. This was a slow process since it had to perform the transformations for each time we created a plot.

We fixed this by using the .persist(memory_option) Spark dataframe command.

```
df.persist(StorageLevel.MEMORY_ONLY)
```

Listing 17: Telling Spark to keep dataframe in memory

After specifying the *Memory_Only* option, the cluster did not have to perform the dataframe transformations for each visualization, since the Spark dataframe was still in memory. This drastically reduced the runtime of our plotting script.

7 RESULTS

In this section we will comment on some of the plots provided. The plots includes three visualizations. From the left we have wordlist scores, then TextBlob and VaderSentiment, and last a scaled average of all three analysers. For the monthly plots we have taken the average for each day and for the year by year plot its for every fourth week. We will also point out some of the main events during each period. More events are available in our visualization notebook in the GitHub repository [11].

7.1 From 2012 to 2022

To get an overview of how the situation has developed we created a visualization (A.4) which illustrates the sentiment scores from the start of the dataset which is 2012 until early 2022. When looking at

[1]Our dataset was around 9,4 million at this time
[2]This was even after we increased the dataset size to 13.4 million tweets

the plot we can see a gradual decrease in sentiment occurring over multiple years.

7.2 February 2014

When looking at plot A.5 we can see that there are three significant spikes. Two which are visible on all and a third which are only visible in TextBlob and Vader, and the scaled average. We did not find any events which could explain the drop on the third. The Maidan Revolution [42] started on the 18th and lasted until the 23th, and we can see some fluctuation leading up to these dates. Lastly the annexation of Crimea started on the 22nd [43], but there are not much change in sentiment here.

7.3 March 2014

For the visualization of March 2014 (A.6) we can see that coming into the month there is a visible drop followed by some stabilization, and when looking at the end of the month we can see that the graph changes more drastically up and down. On the 1th of March Russian legislature approves the use of armed forces [43] which might explain this early drop. Other events that occurred this month was that on the 18th Crimea was formally annexed which seem to result in a downward curve. During the 24th Ukraine orders its troops to withdraw from Crimea. This looks like it leads to a smaller fall followed by a larger one. Last but not least on the 27th the United Nations votes to reject Russian annexation of Crimea [2]. This appears to lead to a considerable increase in positive sentiment.

7.4 June 2014

As for the visualization of March 2014 we can also in June 2014 (A.8) see a visible drop at the start of the month. This is followed by some stability before a vast drop in the middle of the month before stabilising again. At the 14. a Ukrainian air force plane is shot down and 49th people die [44], and on the 16th there is possible disruption for gas to Europe as Russia slash transport to Ukraine [1]. These exact dates the sentiment scores falls significantly which might indicate correlation.

7.5 Beginning of 2022

To get more context we decided to combine our results (A.9) for early 2022. The plots show a steady decrease for both the wordlist implementation and VaderSentiment, and the scaled average, while TextBlob seems to be quite stable through the whole period. Between the 10th and the 14th of January both NATO and the USA hold talks with Russia which lead to no resolution [3][4]. In the same time period there is a increase in negative sentiment which could indicate connection. On the 24th of February Russia launches its invasion of Ukraine [43]. There is a small drop here, but nothing major so its hard to say if it is related. Between the 25th of March to the 28th Russia continues its bombing, but there are also launched major sanctions against Russia such as cutting Russian banks out of SWIFT [7]. There is a serious descend here, but its hard to say if its because of a specific event in this period or just the overall situation.

8 DISCUSSION

Even though not all sentiment analysers use the same scale they still closely follow the same trends. The fact that the results from the analysers usually match increases our beliefs that they are valid and not just random fluctuations. A noticeable difference between VaderSentiment and TextBlob is that TextBlob usually scores a higher positive sentiment than VaderSentiment. It also seems to react less than the other analysers even though comparing to much with the wordlist implementation does not make a lot of sense because of the different scales. One of the reasons behind this, at least when looking at VaderSentiment could be that it is specialised for social media such as Twitter. This could make it better at recognizing words and expressions used in tweets.

Another thing to take into consideration is that while doing preproccessing we removed punctuation's and transformed emojis to text which may have hampered the potential of VaderSentiment. The reason behind this is that VaderSentiment takes emojis and punctuation into account when calculating the sentiment score. However, one could argue that this is not needed because of the large size of the dataset, and that a super accurate sentiment score is not needed when looking at the large scale. We may have gotten more accurate results by personalized data cleaning for each individual sentiment analyser.

9 CONCLUSION

In the introduction to this paper we wanted to evaluate some questions. The answers to these questions are given below:

- **Is it possible to detect major events in the war by looking at time-series plots?** Yes, the results from VaderSentiment and wordlist seems to showcase changes in sentiment scores. TextBlob on the other hand seems to produce scores with low variance centered around zero.
- **Is it possible to detect trends before major events occur?** Yes, in major events such as the Crimea invasion and the Russian invasion in 2022 we think its possible to observe trends on a yearly basis. Although it would be interesting to check this against a control-group to see if this is a global trend, or if it is specific for this case.
- **Is it possible to detect a change in sentiment from before the war until now (April 2022)?** Yes, as mentioned in the previous point there was a decreasing trend from 2012 til the invasion in 2014. After this the sentiment stayed low until a new decrease (starting mid 2021) leading up to the war in 2022.

9.1 Future Work

As mentioned in the discussion; VaderSentiment has the functionality to use both punctuation and emojis, so it would be curious to see if this could improve its results. Another interesting aspect would be to use tweets unrelated to the Ukraine/Russia war as a sort of control group. Lastly in this paper we used a wordlist with 2477 words, so performing analysis with different wordlists would certainly be interesting.

REFERENCES

[1] France 24. 2014. Possible disruption' to Europe as Russia cuts Ukraine gas supply. https://www.france24.com/en/20140616--russia-ukraine-europe-gas-disruption. *Unknown* (2014), 1.

[2] France 24. 2014. UN votes to reject Russian annexation of Crimea. https://www.france24.com/en/20140327-un-general-assembly-votes-reject-russian-annexation-crimea. *Unknown* (2014).

[3] France 24. 2022. OSCE met in Vienna after strained Russia-US, NATO talks. https://www.france24.com/en/video/20220113-osce-met-in-vienna-after-strained-russia-us-nato-talks. *Unknown* (2022).

[4] France 24. 2022. Russia sees no reason to restart talks on Ukraine crisis, threatens action after 'dead end'. https://www.france24.com/en/europe/20220113-russia-sees-no-reason-to-restart-talks-on-ukraine-crisis-threatens-action-after-dead-end. *Unknown* (2022).

[5] Apache. [n. d.]. Standard Scaler. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.StandardScaler.html. Accessed 2022-04-23.

[6] Neal Caren. [n. d.]. Word Lists and Sentiment Analysis. https://nealcaren.org/lessons/wordlists/. Accessed 2022-04-22.

[7] Alan Rappeport David E. Sanger and Matinas Stevis-Gridneff. 2022. The U.S and Europe will bar some Russian banks from SWIFT. https://www.nytimes.com/2022/02/26/us/politics/eu-us-swift-russia.html. *Unknown* (2022).

[8] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. *Unknown* (2004).

[9] Edureka. [n. d.]. Apache Spark with Hadoop – Why it Matters? https://www.edureka.co/blog/apache-spark-with-hadoop-why-it-matters/. Accessed: 2022-04-17.

[10] Data Flair. Not Stated. Apache Hive Architecture – Complete Working of Hive with Hadoop. *Hive Tutorials* (Not Stated).

[11] Ola Andrè Flotve and Håvard Moe Jacobsen. 2022. Twitter Sentiment. https://github.com/havardMoe/Twitter_Sentiment/.

[12] GeeksforGeeks. 2021. Apache Hive Installation With Derby Database And Beeline. https://www.geeksforgeeks.org/apache-hive-installation-with-derby-database-and-beeline/. (2021).

[13] Shubham gosain. [n. d.]. Twitter Sentiment Analysis Using Hadoop. https://github.com/shubhamgosain/twitter-Sentiment-Analysis-using-hadoop. Accessed 2022-04-27.

[14] Hadoop. [n. d.]. Apache Hadoop. https://hadoop.apache.org/. Accessed: 2022-04-19.

[15] Hayley Dorney. [n. d.]. How to create and use hashtags. https://business.twitter.com/en/blog/how-to-create-and-use-hashtags.html. Accessed: 2022-04-18.

[16] Hive. [n. d.]. Hive. https://hive.apache.org/. Accessed 2022-05-01.

[17] Contributors in github repository. [n. d.]. VADER-Sentiment-Analysis. https://github.com/cjhutto/vaderSentiment. Accessed 2022-04-22.

[18] Vladimir Kaplarevic. [n. d.]. How to Install Apache Hive on Ubuntu. https://phoenixnap.com/kb/install-hive-on-ubuntu. Accessed 2022-04-23.

[19] Taehoon Kim and Kevin Wurster. [n. d.]. emoji python library. https://pypi.org/project/emoji/. Accessed 2022-04-23.

[20] matplotlib. [n. d.]. Visualization with Python. https://matplotlib.org/. Accessed 2022-04-23.

[21] MonkeyLearn. [n. d.]. Sentiment Analysis: A Definitive Guide. https://monkeylearn.com/sentiment-analysis/. Accessed: 2022-04-19.

[22] Arun C Murthy, Chris Douglas, Mahadev Konar, Owen O'Malley, Sanjay Radia, Sharad Agarwal, and Vinod KV. 2011. Architecture of next generation apache hadoop mapreduce framework. *Apache Jira* (2011).

[23] F. Å. Nielsen. [n. d.]. AFINN. http://www2.compute.dtu.dk/pubdb/pubs/6010-full.html. Accessed 2022-04-18.

[24] NLTK Project. [n. d.]. Documentation. https://www.nltk.org/. Accessed 2022-04-22.

[25] Paolo Ripamonti. [n. d.]. Twitter Sentiment Analysis. https://www.kaggle.com/code/paoloripamonti/twitter-sentiment-analysis/notebook. Accessed 2022-04-27.

[26] ScienceDirect. [n. d.]. Yet Another Resource Negotiator. https://www.sciencedirect.com/topics/computer-science/yet-another-resource-negotiator. Accessed: 2022-04-19.

[27] Seaborn. [n. d.]. Seaborn docs. https://seaborn.pydata.org/. Accessed 2022-05-01.

[28] Parthvi Shah. [n. d.]. Sentiment Analysis using TextBlob. https://towardsdatascience.com/my-absolute-go-to-for-sentiment-analysis-textblob-3ac3a11d524. Accessed 2022-04-22.

[29] Simplilearn. [n. d.]. What Is Apache Hive? https://www.youtube.com/watch?v=ynNFghq-zdE&ab_channel=Simplilearn. Accessed: 2022-04-20.

[30] Simplilearn. [n. d.]. What is Hive?: Introduction To Hive in Hadoop. https://www.simplilearn.com/what-is-hive-article#what_is_hive_in_hadoop. Accessed: 2022-04-18.

[31] Simplilearn. [n. d.]. YARN Tutorial. https://www.youtube.com/watch?v=KqaPMCMHH4g. Accessed: 2022-04-20.

[32] Spark. [n. d.]. Apache Spark. https://spark.apache.org/. Accessed: 2022-04-20.

[33] Spark. [n. d.]. Apache Spark. https://spark.apache.org/downloads.html. Accessed: 2022-04-25.

[34] Apache Spark. [n. d.]. Running Spark on YARN. https://spark.apache.org/docs/latest/running-on-yarn.html. Accessed: 2022-04-21.

[35] Statista Research Department. [n. d.]. Most popular social networks worldwide as of January 2022, ranked by number of monthly active users. https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/. Accessed: 2022-04-18.

[36] TextBlob. [n. d.]. TextBlob: Simplified Text Processing. https://textblob.readthedocs.io/en/dev/. Accessed: 2022-04-21.

[37] Tweepy. [n. d.]. Tweepy. https://www.tweepy.org/. Accessed: 2022-04-18.

[38] Twitter. [n. d.]. Full-archive search. https://developer.twitter.com/en/docs/twitter-api. Accessed 2022-04-22.

[39] Twitter. [n. d.]. Full-archive search. https://developer.twitter.com/en/docs/twitter-api/tweets/search/introduction. Accessed 2022-04-22.

[40] Twitter. [n. d.]. Search Tweets API. https://developer.twitter.com/en/docs/twitter-api/tweets/search/introduction. Accessed: 2022-04-19.

[41] Twitter Documentation. [n. d.]. Coutning Characters. https://developer.twitter.com/en/docs/counting-characters. Accessed: 2022-04-18.

[42] Wikipedia. [n. d.]. Revolution of Dignity. https://en.wikipedia.org/wiki/Revolution_of_Dignity. Accessed 2022-04-26.

[43] Wikipedia. [n. d.]. Russo-Ukrainian War. https://en.wikipedia.org/wiki/War_in_Donbas. Accessed 2022-04-26.

[44] Wikipedia. [n. d.]. War in Donbas. https://en.wikipedia.org/wiki/War_in_Donbas. Accessed 2022-04-26.

[45] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*. Unkown.

A GRAPHS AND FIGURES

A.1 Github repository and Dataset

Our code and plots are available at our GitHub repository [11].

Completed Applications (2)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20220427204620-0001	wordlist_MR_analysis	12	1024.0 MiB		2022/04/27 20:46:20	ubuntu	FINISHED	2.3 min
app-20220427204514-0000	wordlist_DF_analysis	12	1024.0 MiB		2022/04/27 20:45:14	ubuntu	FINISHED	1.0 min

Figure A.1: Runtime comparison of MR and WC implementations

app-20220422202038-0001	pre_processing	12	1024.0 MiB		2022/04/22 20:20:38	ubuntu	FINISHED	3.9 min
app-20220422201937-0000	DB_setup	12	1024.0 MiB		2022/04/22 20:19:37	ubuntu	FINISHED	55 s

Figure A.2: Runtime of cleaning and setup

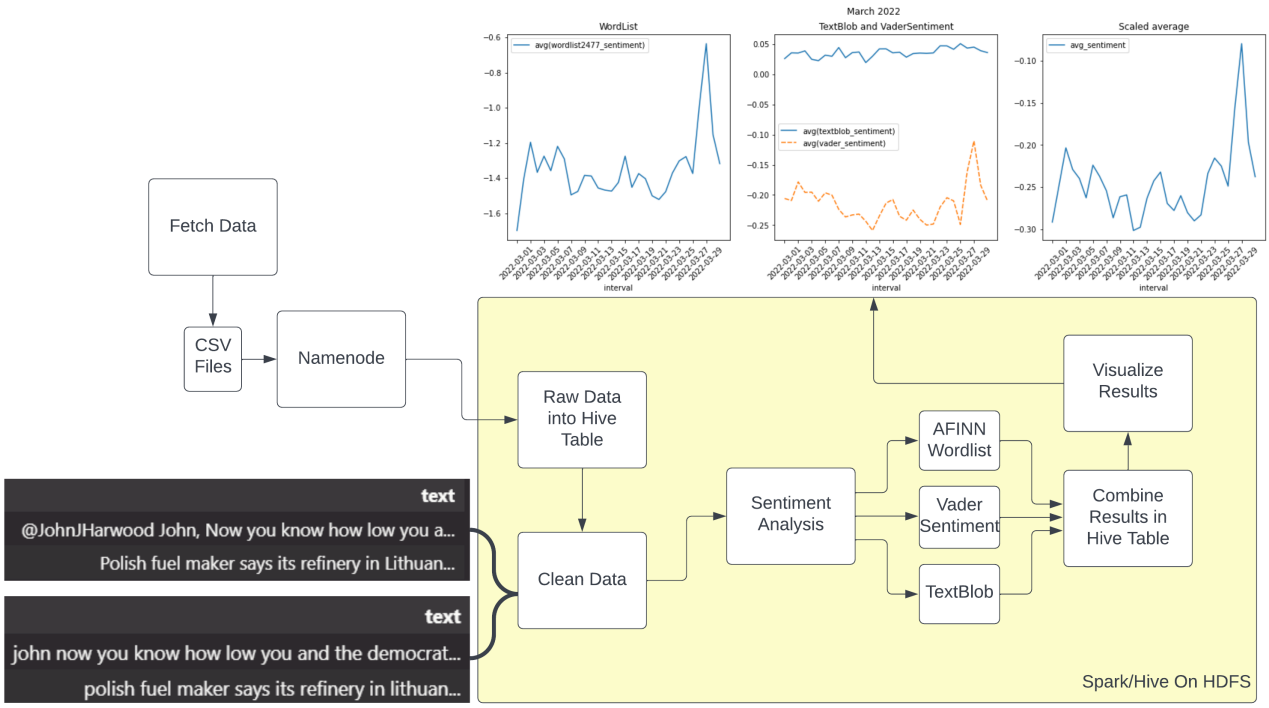

Figure A.3: Complete Pipeline

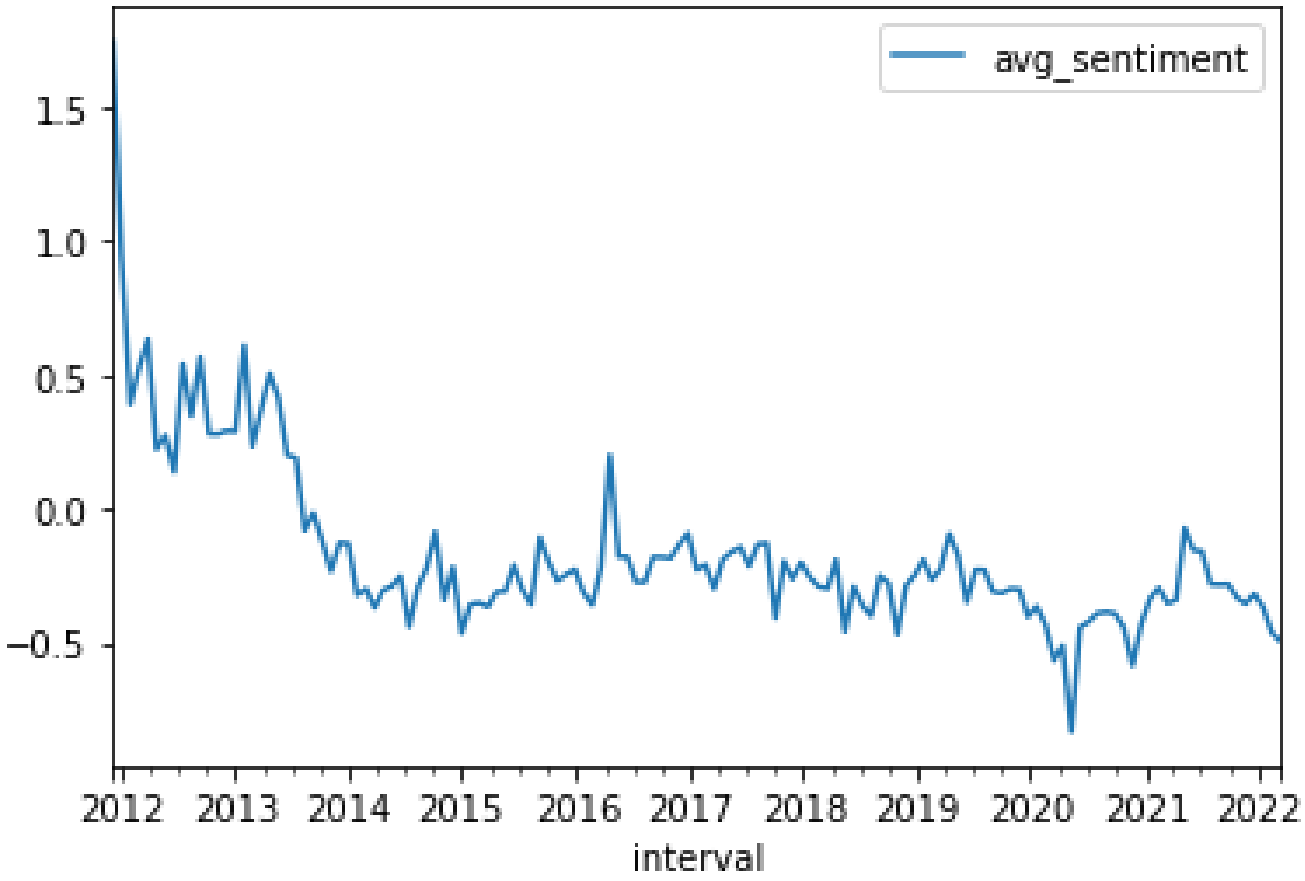

Figure A.4: January 2012 - April 2022

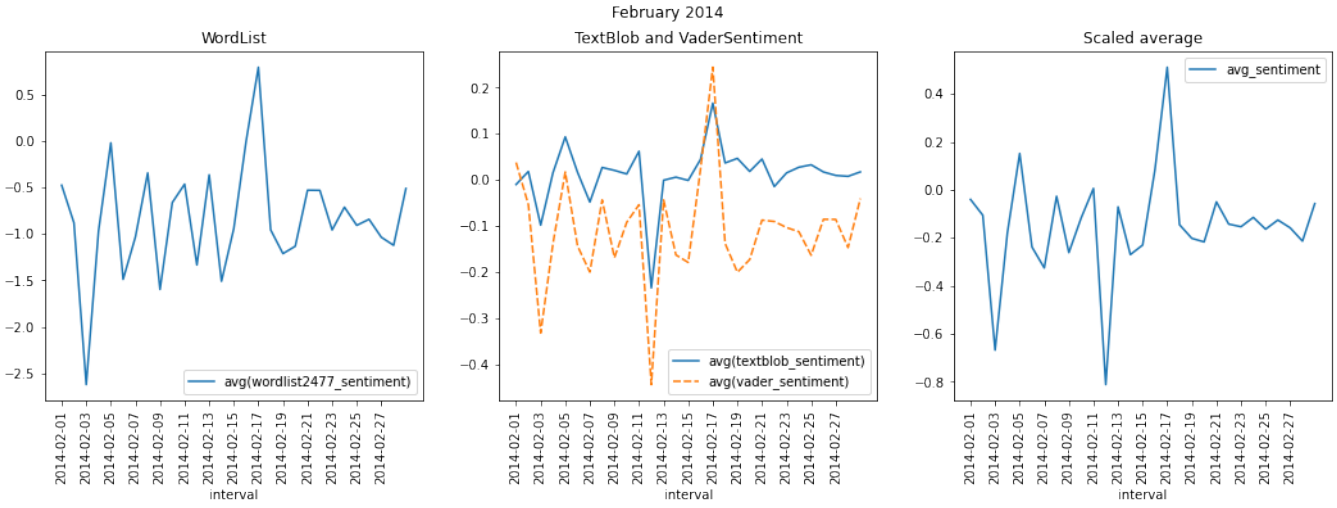

Figure A.5: February 2014

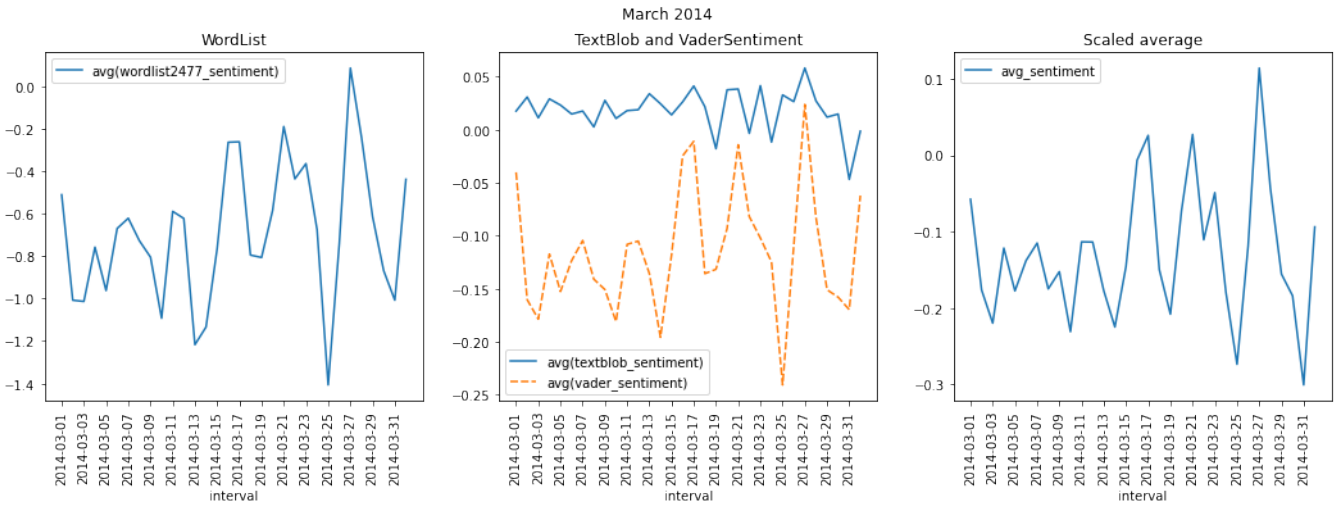

Figure A.6: March 2014

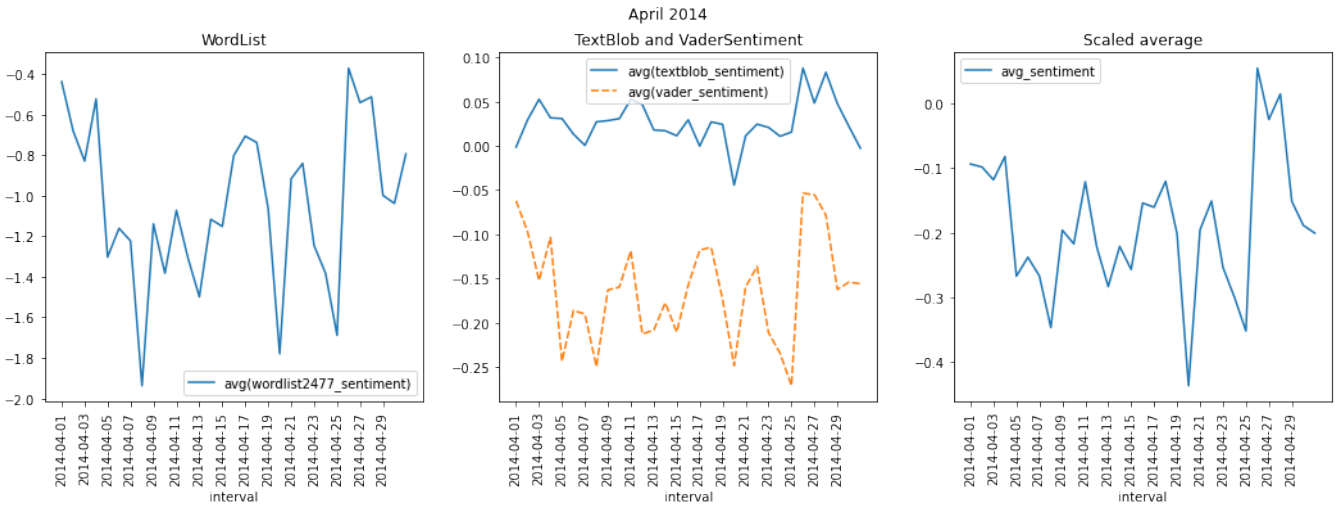

Figure A.7: April 2014

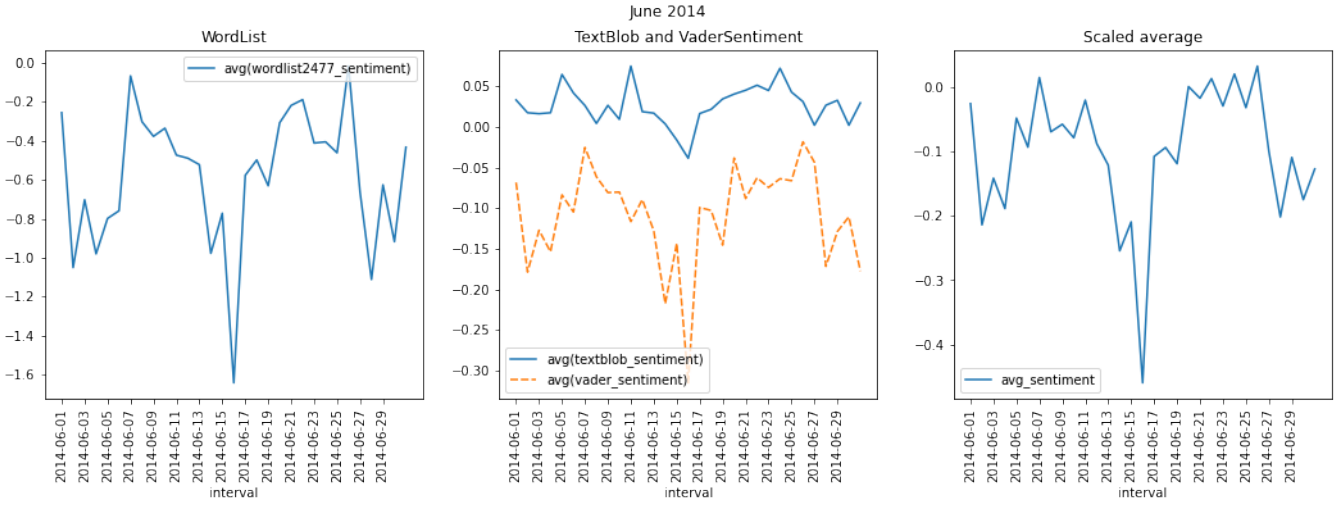

Figure A.8: June 2014

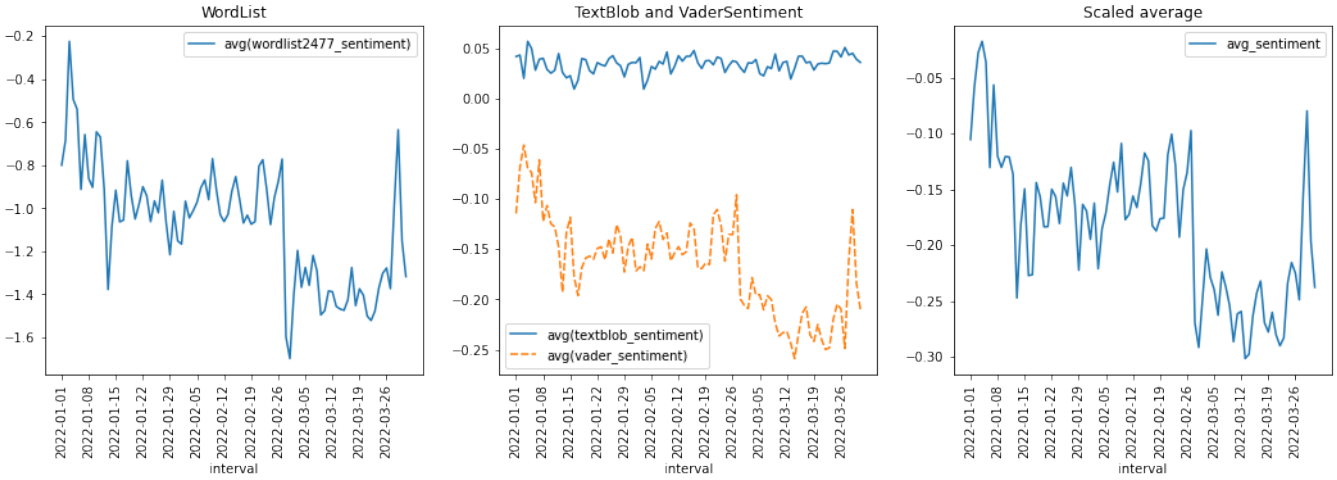

Figure A.9: January-March 2022