Semi-Autonomous Trend Detection Using Real-Time Twitter Data

Final Report (Semester 2)

Danny O'Leary

20067817

Panel 3

Supervisor: Peter Carew

BSc(Hons) in Computer Forensics and Security

Contents

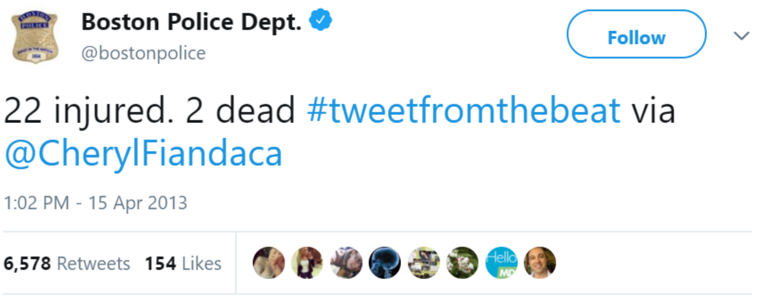
1. Introduction

Setup information can be found at: https://github.com/dannyoleary1/Final-year-project---Twitter-Analysis-using-Machine-Learning

1.1 Background

In Recent years, Twitter, and other social media sites have been growing dramatically. Hundreds of millions of people are using these sites every day, and it has grown its practicality over time. Twitter for example, has become a great place for users to stay up to date with the latest news, and many news sites use Twitter to spread their latest news, for example The Guardian has now amassed over 7 million followers [1].

Not only has Twitter become a great place to catch up with news, but it also has become one of the best places for breaking news. The reason that Twitter has become a great place for breaking news is partly because it has over 500 million tweets every day with over 330 million active users [2]. This means that a Twitter user could very well be a first-hand source of an event, and since they can simply just Tweet about it from a mobile device, it can very quickly gain in popularity. The difference with this over a news site would be that a Publisher is needed, and the account may very well not be first-hand. An example of this was the Boston Marathon bombing that occurred on April 15th, 2013. Within seconds of the bombings taking place, there was first-hand reports in the form of Tweets which included text, images, and videos. It was a further 15 minutes before the incident was reported on any major news station on tv, and even longer before it was reported on their websites. Along with this, it is interesting to note that the Boston Police also provided updates via Twitter [3].

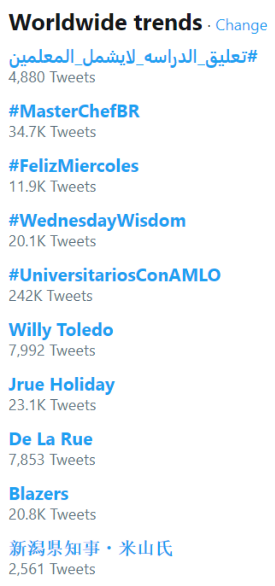

Boston Police Dept.
@bostonpolice

22 injured. 2 dead #tweetfromthebeat via @CherylFiandaca
1:02 PM - 15 Apr 2013
6,578 Retweets 154 Likes

Figure 1. One of the Tweets from the Boston Police that was tweeted on the day [4].

One of the ways that Twitter deals with '*breaking news*' on a global level is by using trend detection. This means that Twitter is searching for patterns in their data to suggest that something is gaining in popularity. This is useful for Twitter to keep people '*in the loop*' about the latest Twitter news. A question that could be asked is '*why trend detection is important?'*. To answer this question, it is important to realize that trend detection in its basic sense is just finding something that is increasing in popularity. Why would you want to know if something is increasing in popularity? This is important for several people:

1. News Organizations: News Organizations want to try being the first to try and break a story. By using trend detection, it could help to spot emerging stories as they happen.
2. Advertisers: Advertisers are interested in trend detection so that they can target their advertising around what is becoming popular. This may lead to more successful target marketing for the advertisers based on knowing more information.
3. First Responders: First Responders are interested in knowing what they are arriving at so that they can plan accordingly. Trend detection is good for this because it can give accounts of people who are at the scene often with photos or video clips.



Worldwide trends · Change

#تعليق_الدراسه_لايشمل_المعلمين
4,880 Tweets

#MasterChefBR
34.7K Tweets

#FelizMiercoles
11.9K Tweets

#WednesdayWisdom
20.1K Tweets

#UniversitariosConAMLO
242K Tweets

Willy Toledo
7,992 Tweets

Jrue Holiday
23.1K Tweets

De La Rue
7,853 Tweets

Blazers
20.8K Tweets

新潟県知事・米山氏
2,561 Tweets

Figure 2. This is an example of Twitters trending topics.

Having knowledge of trends tells us a lot of information such as what people are attracted to, and what people think is currently important [5].

1.2 What this project is

In section '*1.1 Background*' who trend detection is important to was discussed, and 3 different jobs which would be interested were talked about: first responders, advertisers, and news organizations. This is true for the way that Twitter does categories, but for example wouldn't a security company be interested in detecting security issues that are happening in real time?

The way Twitter has done trends is fantastic for what they are trying to achieve. However, it has some drawbacks. Twitter breaks it down into 3 different categories: global trends, local trends, and tailored trends. Each of these categories has only the 10 most popular trends for that given time. It is important to talk about how the tailored trends work because that is what this project is closely based off. The aim of tailored trends is to give trends based on your location and who you follow. This causes a problem when you may not be interested in a lot of the topics that the people you are following are interested in e.g. You follow a person for their opinion on football, but they also tweet a lot about fishing which you have no interest in. This makes it so that a lot of the tailored trends that are being detected are not relevant to you.

Another issue with the way Twitter does trends is the fact it only has 10 entries and isn't followed over time. Following a story over time can lead to a more accurate representation of what it is that is going on.

The idea behind this project is to develop a web application which a user can use to detect trends on the topics they are interested in. The web app will work in a semi-autonomous way where it will do detection of trends in the background. It will detect trends in real time. The trend can be monitored over time and isn't removed until the user wants it removed. This opens the usefulness open to a much different category of people.

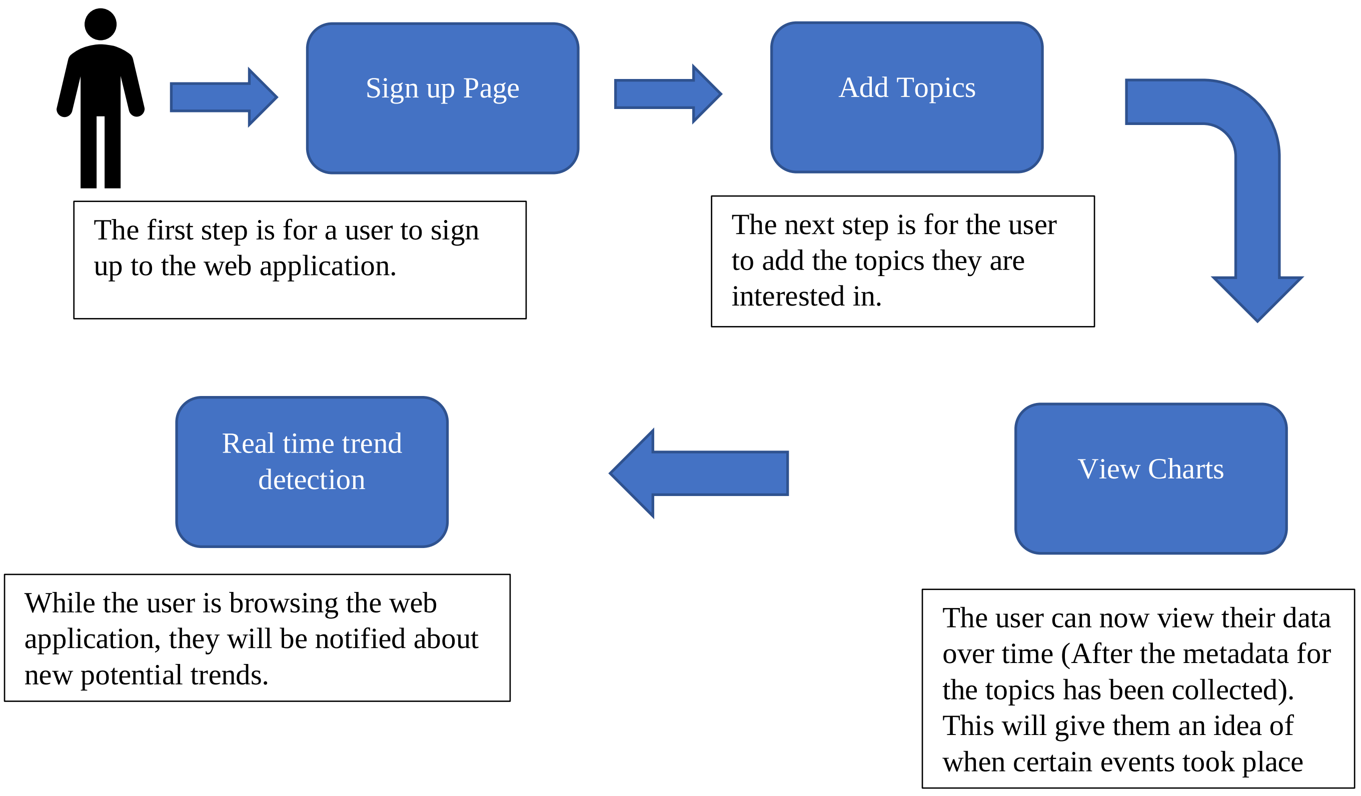
Some of the features of the web application include:

1. Semi-Autonomous trend detection on real time Twitter data.
2. Adding the topics to detect trends for based on the user's preference.
3. Adding followers/Twitter users to detect topics that you may wish to include. A user of the web application can add Twitter users that they are interested in to help with picking the right topics for them. This feature will report back a list of potential topics which can then be added/declined.
4. Collect old tweets based on a topic to detect what some of the keywords on a topic were. An example of this would be to collect tweets on 'youtube' after it has had an outage. This feature will return the most popular words based on this, which may be: 'down', 'outage', or 'hacked'. This gives a user a suggestion on what topics may be useful to get a broad range of trends e.g. outage would be a good topic to include if you were interested in telling when websites go down.
5. Table views. This is a table view of the data that has been collected over time with the potential trends displayed in red. The table can be changed based on the topic you want.
6. Display information on charts. The charts that are currently supported are: spline, line, bar, area, and area spline.

2. Project Use Case

A sample of how the user would interact with this project could be as follows:

2.1 Basic Use Case



The first step is for a user to sign up to the web application.

The next step is for the user to add the topics they are interested in.

The user can now view their data over time (After the metadata for the topics has been collected). This will give them an idea of when certain events took place

While the user is browsing the web application, they will be notified about new potential trends.

2.2 More complete use case using the Web Application

This example will go through the use case using figures from the actual web application. In this scenario the user has already signed up to the web application. The person using the web application is interested in detecting trends to do with Security. The reason for this is because the person wants to be prepared about the latest threats that could affect his business.

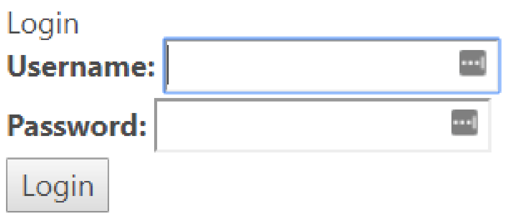


Figure 3. First the user logs in to the Web Application.

Now the user is redirected to the dashboard page.

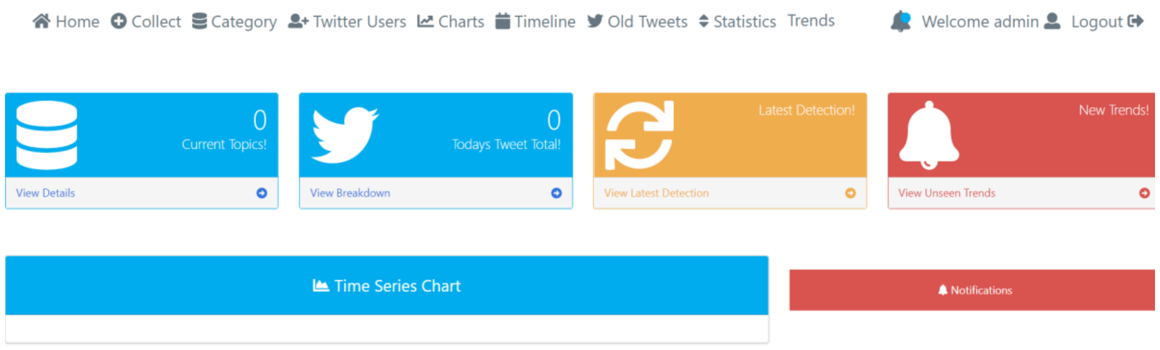


Figure 4. An example of what the dashboard would look like for this user.

The user isn't sure what keywords they are looking for so they decide to use 2 of the features provided on the web application: '*Twitter Users*' which will allow them to enter Twitter users they are interested in finding keywords from, and also the '*Old Tweets*' section which will be useful to search old security issues to find out what was being Tweeted about during that time.

The user comes up with a list of security professionals that they want to come up with Topics from:

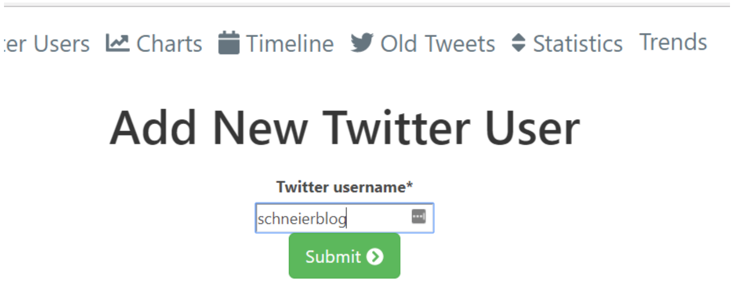
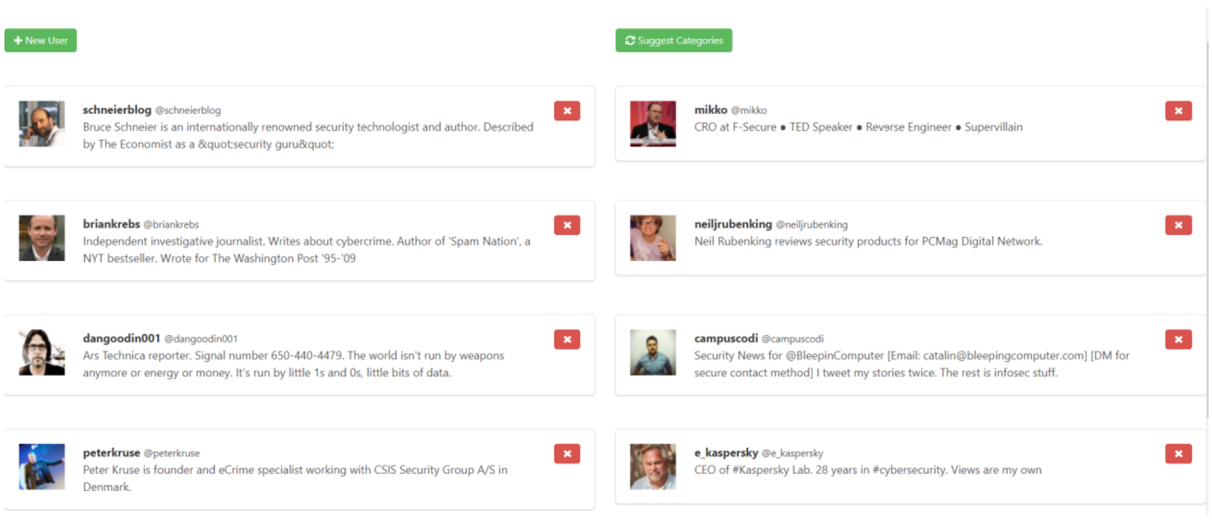


Figure 5. This is the page where a user can add Twitter users.



schneierblog @schneierblog
Bruce Schneier is an internationally renowned security technologist and author. Described by The Economist as a &quot;security guru&quot;

mikko @mikko
CRO at F-Secure ● TED Speaker ● Reverse Engineer ● Supervillain

briankrebs @briankrebs
Independent investigative journalist. Writes about cybercrime. Author of 'Spam Nation', a NYT bestseller. Wrote for The Washington Post '95-'09

neiljrubenking @neiljrubenking
Neil Rubenking reviews security products for PCMag Digital Network.

dangoodin001 @dangoodin001
Ars Technica reporter. Signal number 650-440-4479. The world isn't run by weapons anymore or energy or money. It's run by little 1s and 0s, little bits of data.

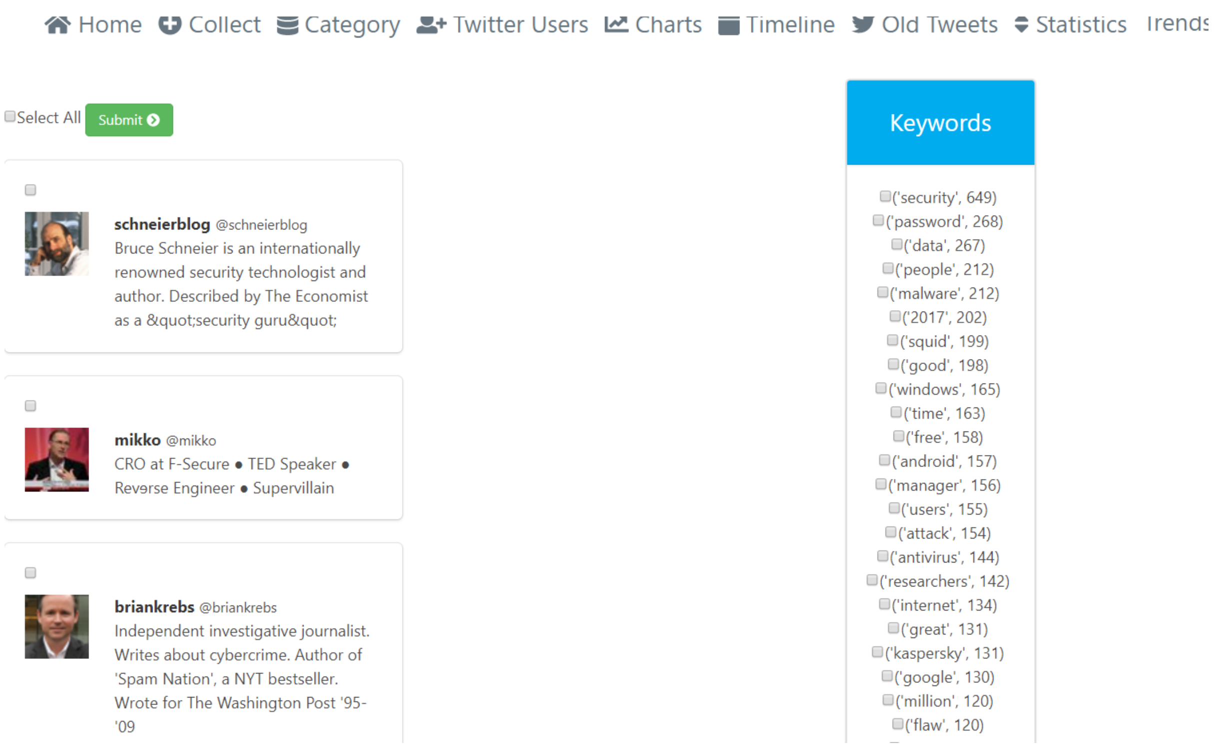
campuscodi @campuscodi
Security News for @BleepinComputer [Email: catalin@bleepingcomputer.com] [DM for secure contact method] I tweet my stories twice. The rest is infosec stuff.

peterkruse @peterkruse
Peter Kruse is founder and eCrime specialist working with CSIS Security Group A/S in Denmark.

e_kaspersky @e_kaspersky
CEO of #Kaspersky Lab. 28 years in #cybersecurity. Views are my own

Figure 6. This is the page where the Twitter users will be added. This is the result for this example.

Now the user will click on the suggest categories button to come up with some keywords.



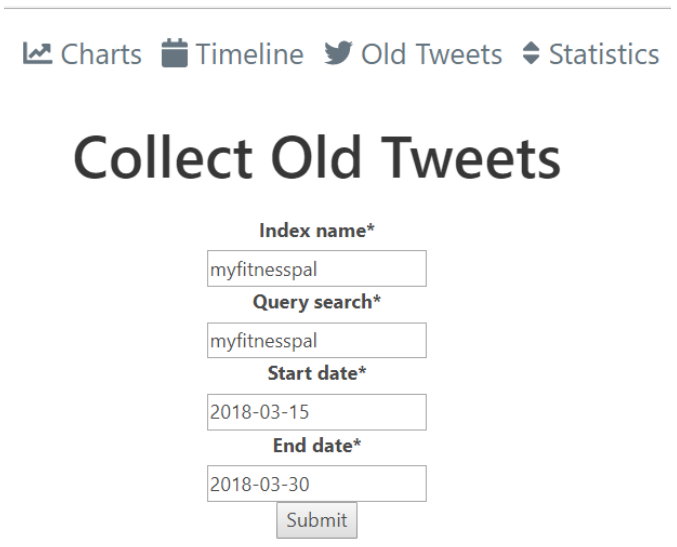
schneierblog @schneierblog
Bruce Schneier is an internationally renowned security technologist and author. Described by The Economist as a &quot;security guru&quot;

mikko @mikko
CRO at F-Secure ● TED Speaker ● Reverse Engineer ● Supervillain

briankrebs @briankrebs
Independent investigative journalist. Writes about cybercrime. Author of 'Spam Nation', a NYT bestseller. Wrote for The Washington Post '95-'09

('security', 649)
('password', 268)
('data', 267)
('people', 212)
('malware', 212)
('2017', 202)
('squid', 199)
('good', 198)
('windows', 165)
('time', 163)
('free', 158)
('android', 157)
('manager', 156)
('users', 155)
('attack', 154)
('antivirus', 144)
('researchers', 142)
('internet', 134)
('great', 131)
('kaspersky', 131)
('google', 130)
('million', 120)
('flaw', 120)

Figure 7. The user selects all the Twitter users that were just added and then the keywords are displayed. The user can now select which ones they want to use for topics. In this case, security, password, and data are chosen.



Collect Old Tweets

Index name*: myfitnesspal

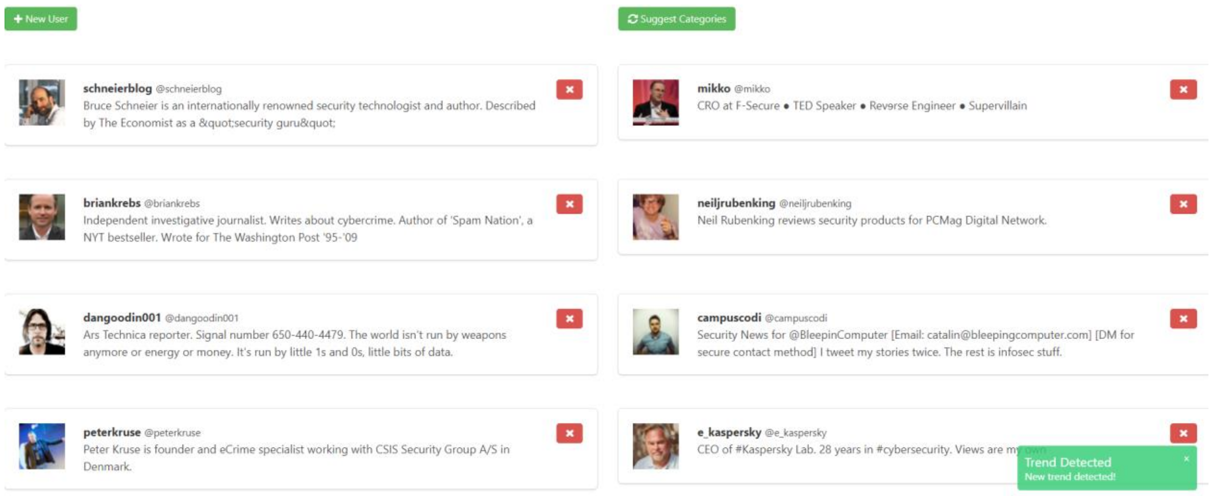
Query search*: myfitnesspal

Start date*: 2018-03-15

End date*: 2018-03-30

Figure 8. Next the user has some recent security issues that may contain relevant keywords. The user finds another few topics to add in from this: bug, hacked, and breach.

Now the user is setup to wait for real time trend detection.



schneierblog @schneierblog
Bruce Schneier is an internationally renowned security technologist and author. Described by The Economist as a &quot;security guru&quot;

mikko @mikko
CRO at F-Secure • TED Speaker • Reverse Engineer • Supervillain

briankrebs @briankrebs
Independent investigative journalist. Writes about cybercrime. Author of 'Spam Nation', a NYT bestseller. Wrote for The Washington Post '95-'09

neiljrubenking @neiljrubenking
Neil Rubenking reviews security products for PCMag Digital Network.

dangoodin001 @dangoodin001
Ars Technica reporter. Signal number 650-440-4479. The world isn't run by weapons anymore or energy or money. It's run by little 1s and 0s, little bits of data.

campuscodi @campuscodi
Security News for @BleepinComputer [Email: catalin@bleepingcomputer.com] [DM for secure contact method] I tweet my stories twice. The rest is infosec stuff.

peterkruse @peterkruse
Peter Kruse is founder and eCrime specialist working with CSIS Security Group A/S in Denmark.

e_kaspersky @e_kaspersky
CEO of #Kaspersky Lab. 28 years in #cybersecurity. Views are my own

Trend Detected
New trend detected!

Figure 9. A notification appears in the bottom right of the screen that notifies the user about a new trend being detected.

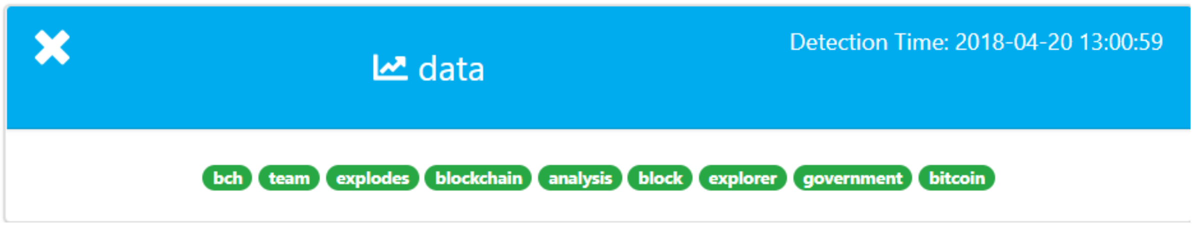
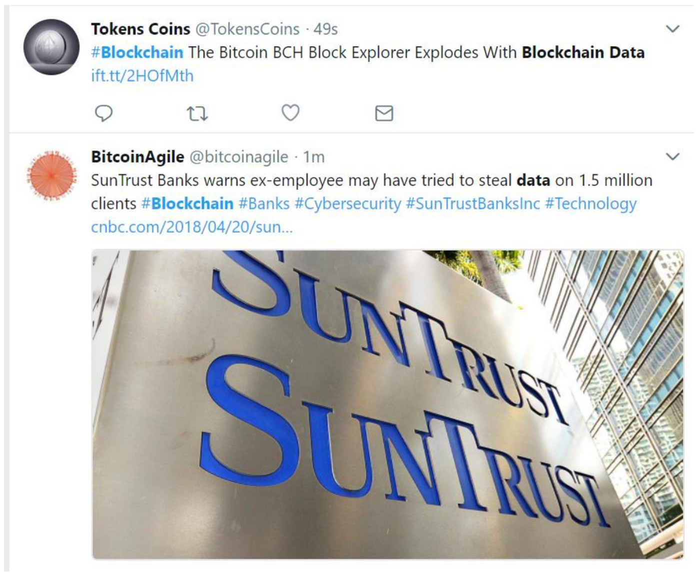
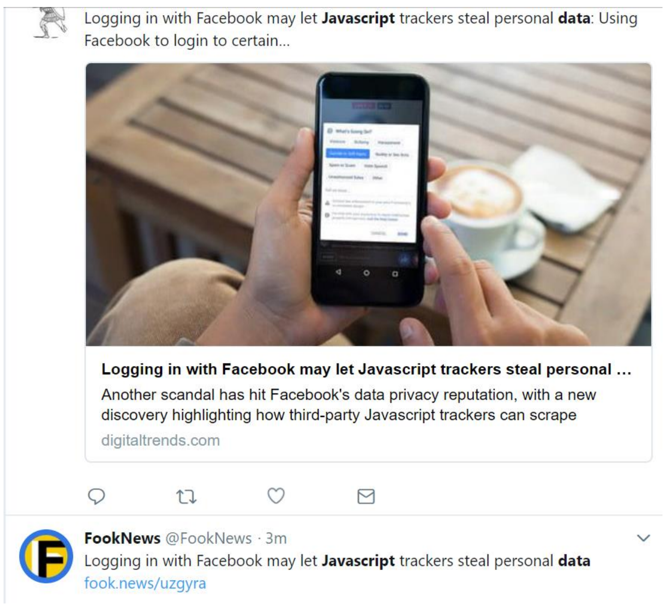


Figure 10. It shows a breakdown of the keywords that have just been picked up. If the user clicks on the word, it will redirect to Twitter and the latest results.


Tokens Coins @TokensCoins · 49s
#**Blockchain** The Bitcoin BCH Block Explorer Explodes With **Blockchain Data**
ift.tt/2HOfMth

BitcoinAgile @bitcoinagile · 1m
SunTrust Banks warns ex-employee may have tried to steal **data** on 1.5 million clients #**Blockchain** #Banks #Cybersecurity #SunTrustBanksInc #Technology
cnbc.com/2018/04/20/sun...

Figure 11. After clicking on the keyword blockchain, it shows a news story that has just been posted from cnbc about a potential data breach.


Logging in with Facebook may let **Javascript** trackers steal personal **data**: Using Facebook to login to certain...

Logging in with Facebook may let Javascript trackers steal personal ...
Another scandal has hit Facebook's data privacy reputation, with a new discovery highlighting how third-party Javascript trackers can scrape
digitaltrends.com

FookNews @FookNews · 3m
Logging in with Facebook may let **Javascript** trackers steal personal **data**
fook.news/uzgyra

Figure 12. Another example of a potential vulnerability after clicking on JavaScript.

The last thing that a user can do is then get a breakdown of the what the topics look like over time. This is good for spotting increases over days/weeks.

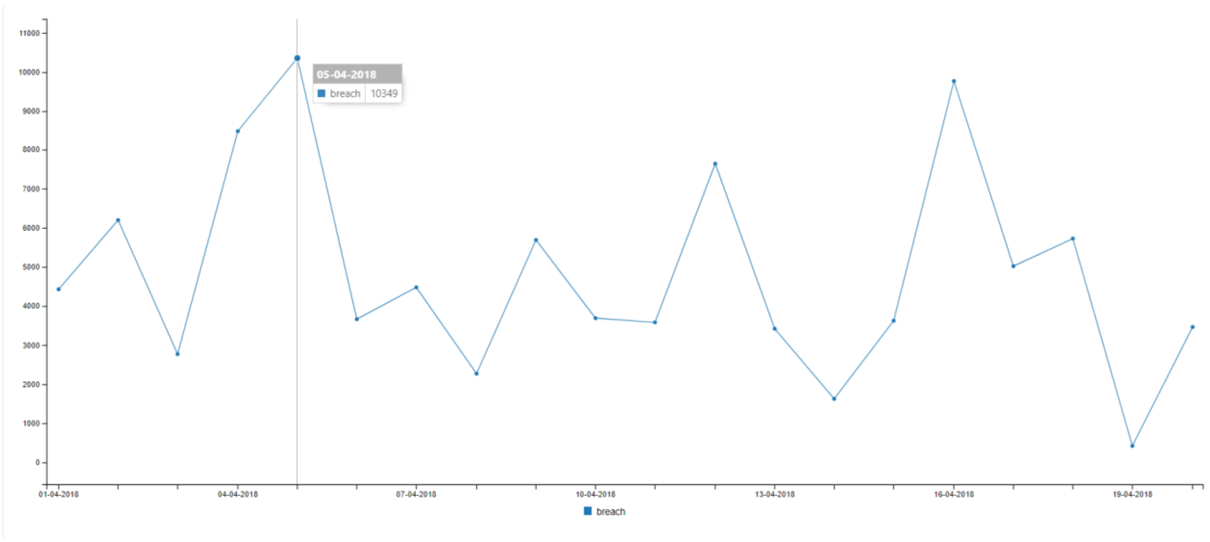
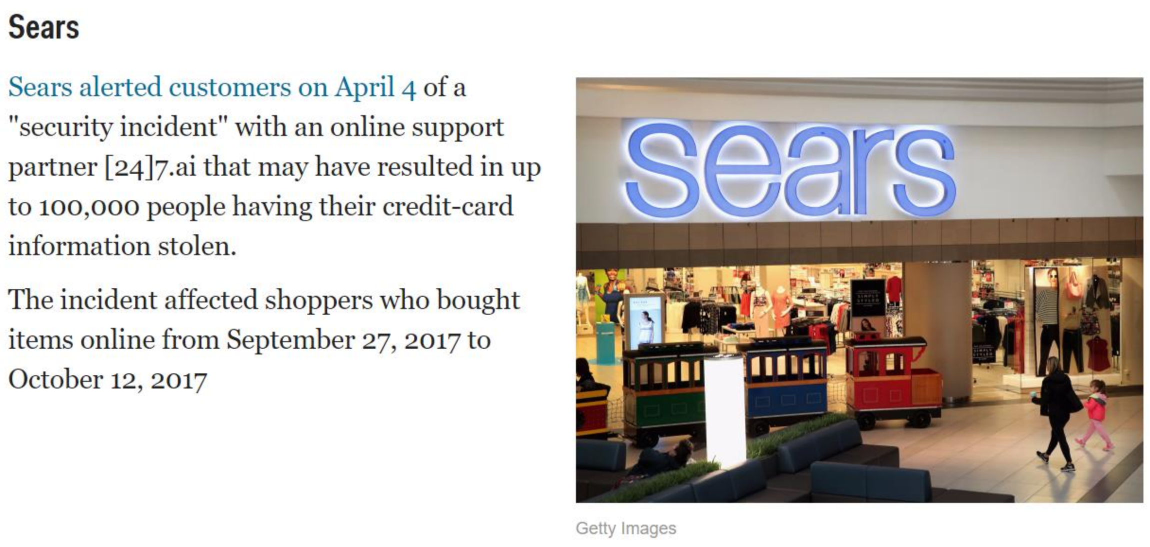


Figure 13. An example of a chart for the topic 'breach'. As can be seen from the chart, something probably happened on the 4th of April and lead into the 5th of April as seen by the massive spike in the data.



Sears

Sears alerted customers on April 4 of a "security incident" with an online support partner [24]7.ai that may have resulted in up to 100,000 people having their credit-card information stolen.

The incident affected shoppers who bought items online from September 27, 2017 to October 12, 2017

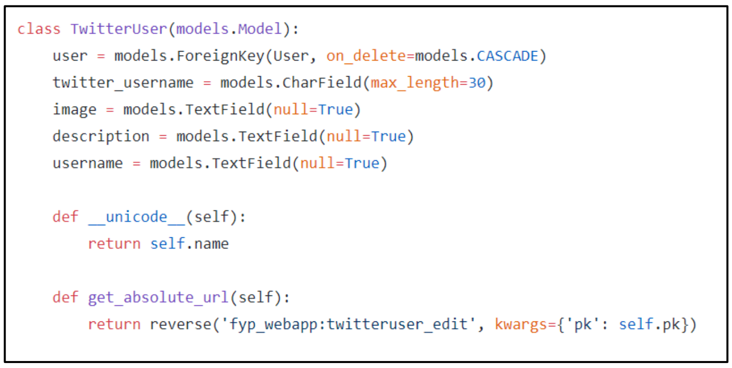
Figure 14. An example of what the breach for that specific date probably was [8].

3. Database Storage – SQLite and Elasticsearch

The databases that are being used in this project include both a SQLite database that is in built to Django, and Elasticsearch will be used to store the Twitter data that's used. The reason for having 2 separate databases for storage has to do with the data that is being stored. A lot of the time the SQLite database will be used will be because it's easier than doing the same in Elasticsearch. However, Elasticsearch is still the main database that is being used in the project because of its custom search queries which mean all data doesn't need to be loaded into memory at once, it's schema free, and is quicker to return large quantities of information.

3.1 SQLite

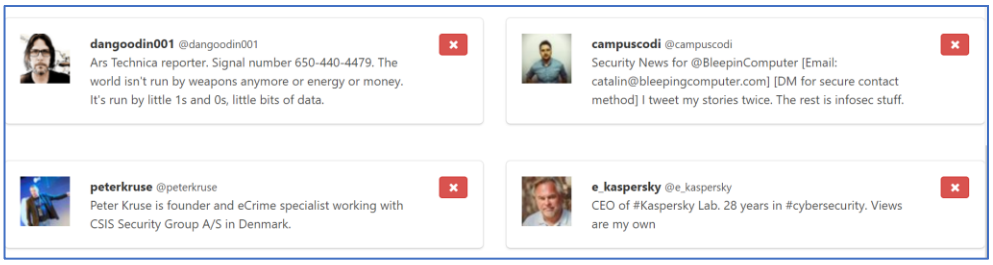
Django uses a SQLite database by default. This means that no set up is needed for this database since Python already has SQLite included. SQLite is a relational database, and it will be used to store Django models. Django models are basically objects that are responsible for representing your data. They contain the fields that are going to be used to create the Database schema.



```python
class TwitterUser(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    twitter_username = models.CharField(max_length=30)
    image = models.TextField(null=True)
    description = models.TextField(null=True)
    username = models.TextField(null=True)

    def __unicode__(self):
        return self.name

    def get_absolute_url(self):
        return reverse('fyp_webapp:twitteruser_edit', kwargs={'pk': self.pk})
```

Figure 15. This is an example of one of the models used. This one is responsible for saving Twitter Users to a database.

As can be seen in figure 15 there's several fields that are needed. The user field is what links the objects to a specific user. The *'twitter_username'* field is the username that is being added to the system. The image, description, and username are just for styling the information on the front-end. None of this is typed in by the user and is dynamically set within the code.



dangoodin001 @dangoodin001
Ars Technica reporter. Signal number 650-440-4479. The world isn't run by weapons anymore or energy or money. It's run by little 1s and 0s, little bits of data.

campuscodi @campuscodi
Security News for @BleepinComputer [Email: catalin@bleepingcomputer.com] [DM for secure contact method] I tweet my stories twice. The rest is infosec stuff.

peterkruse @peterkruse
Peter Kruse is founder and eCrime specialist working with CSIS Security Group A/S in Denmark.

e_kaspersky @e_kaspersky
CEO of #Kaspersky Lab. 28 years in #cybersecurity. Views are my own

Figure 16. A sample of what the result looks like on the front end.

The models that are in this project are as follows:

1. User model: The user model is controlled by Django authentication and is a secure way of authenticating a user. This model will be responsible for dealing with logins and giving access to the rest of the database entries.
2. Topic Model: The topic model is responsible for the information about an added topic. It is a rather simple model that is connected to a logged in user. It has only one field, the topic name.
3. Twitter User Model: The model discussed in figure 15. It allows a user to add a Twitter user they are interested in collecting keywords for.
4. Notification Tracked Model: This model is what is used to keep track of trends over time. Both new trends and already saved ones are kept here. It has two fields, the topic and the keywords. The keywords will have the words that are appearing lots of times (or the trend), and the topic will be the topic that it is responsible for.

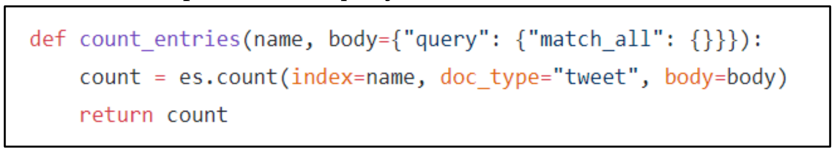
3.2 Elasticsearch

Elasticsearch is used a lot in this project and is one of the major foundations that the project is built on. One of the first things to note is that this project contains many different custom queries to the Elasticsearch database. This means that the project contains a custom API for Elasticsearch calls using the Elasticsearch-py library [6]. One of the most useful features of this library for this project is the scroll() function. This function allows for large quantities of data to be '*scrolled*' through. This means that data wont all be loaded at once and instead it will be incremented as you go. This will be extremely useful for iterating through the latest results of tweets. The way Elasticsearch works on this project is that each topic has 3 indexes each (This will be discussed below). Indexes are like databases. This is where information will be stored. An example of an index in the link '*api/entry/1*' would be '*api*'.

3.2.1 API Calls/Utility Functions

The API calls that are included in this web application are:

1. **Create index**. A straightforward call to create an index. It reads in a String which will be used for the index name.
2. **Check if index exists**. This will return true/false based on if the index is already in elasticsearch. It reads in a String and compares it to what already exists.
3. **Delete index**. It reads in a String and deletes that index if it exists.
4. **List all entries**. This will return a list of all the indexes that can be found in elasticsearch.
5. **Count entries**. This will read in a String and return how many documents(entries) are in that index. Note that this is setup so that the query can be overridden elsewhere in the code.

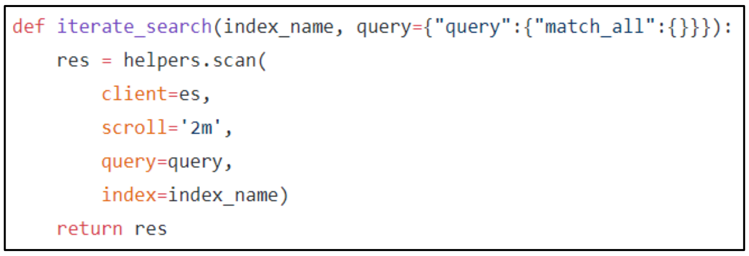


```
def count_entries(name, body={"query": {"match_all": {}}}):
    count = es.count(index=name, doc_type="tweet", body=body)
    return count
```

Figure 17. The body query can be changed so maybe the application only wants the count for the number of times a certain tweet appears. To do this, it would just read in a different query for the body.

6. **Add entry**. Reads in JSON Data and aggregates it to a supplied String for the index name.
7. **Delete entry**. Deletes an entry based on an id, and an index name.

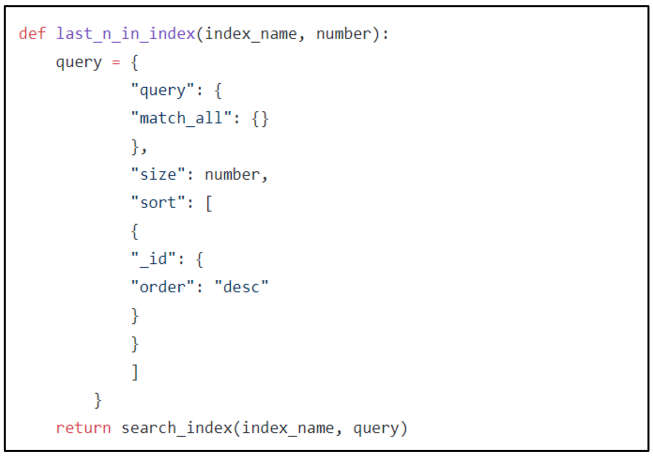
8. **Search index**. This will take in an index, and a query and return several documents. Note that this shouldn't be used for results that are expected to be substantial in size. Like number 5, this can also be overwritten in the same way.
9. **Iterate search**. This is much like the '*search index*' function. The only difference is that this will iterate through the results as it goes and save memory. This is the preferred choice for larger documents to prevent crashes. This function does run slower than the '*search index*' function. It can also be overwritten like the '*search index*' function. Note that the time this stays open for is currently 2 minutes which can also be changed. This is to prevent the search from not being closed properly.



```
def iterate_search(index_name, query={"query":{"match_all":{}}}):
    res = helpers.scan(
        client=es,
        scroll='2m',
        query=query,
        index=index_name)
    return res
```

Figure 18. An example of the iterate_search function.

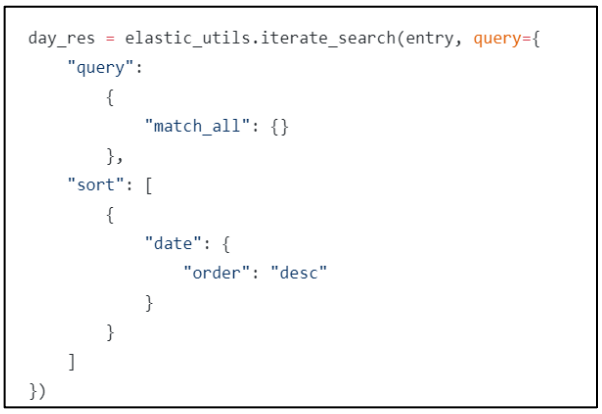
10. **Last 'n' in index**. This is a function that will return 'n' number of results based on a passed in number.



```
def last_n_in_index(index_name, number):
    query = {
            "query": {
            "match_all": {}
            },
            "size": number,
            "sort": [
            {
            "_id": {
            "order": "desc"
            }
            }
            ]
        }
    return search_index(index_name, query)
```

Figure 19. The last n in index function.

Along with these utility functions/API calls custom queries are used throughout the project for different reasons.



```
day_res = elastic_utils.iterate_search(entry, query={
    "query":
        {
            "match_all": {}
        },
    "sort": [
        {
            "date": {
                "order": "desc"
            }
        }
    ]
})
```

Figure 20. An example of a custom query being used to get an index's results by date.

3.2.2 How tweets are stored

Topics are broken down into 3 separate indexes. Below is an example of how this works along with a practical example for the topic '*security*':

Use case:

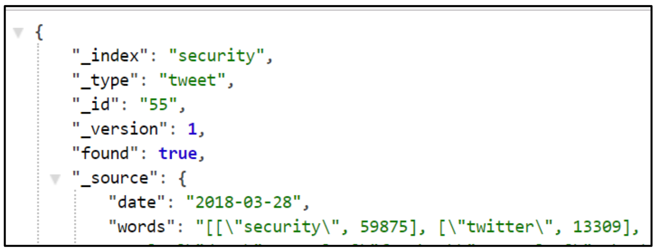
topic Index
topic-latest Index
topic-median Index

Practical example:

security
security-latest
security-median

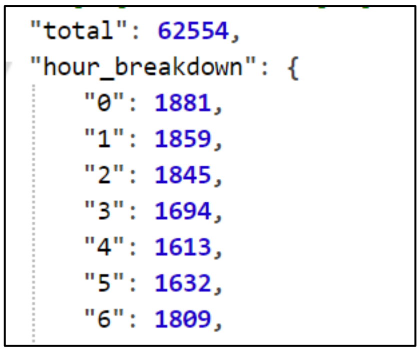
3.2.2.1 What the 'topic index' stores

The topic index is responsible for storing the day to day results. This is basically metadata about the Tweeting patterns for that day. The information that it stores are: the date, an hour breakdown of how many tweets occurred during that day, the total number of tweets for that date, and the most common 350 words on that topic (Also includes the count for that specific word).



```
{
    "_index": "security",
    "_type": "tweet",
    "_id": "55",
    "_version": 1,
    "found": true,
    "_source": {
        "date": "2018-03-28",
        "words": "[[\"security\", 59875], [\"twitter\", 13309],
```

Figure 21. This is an example of the entries that are stored on one part of this index. This image includes the captured date, and the start of the words breakdown.



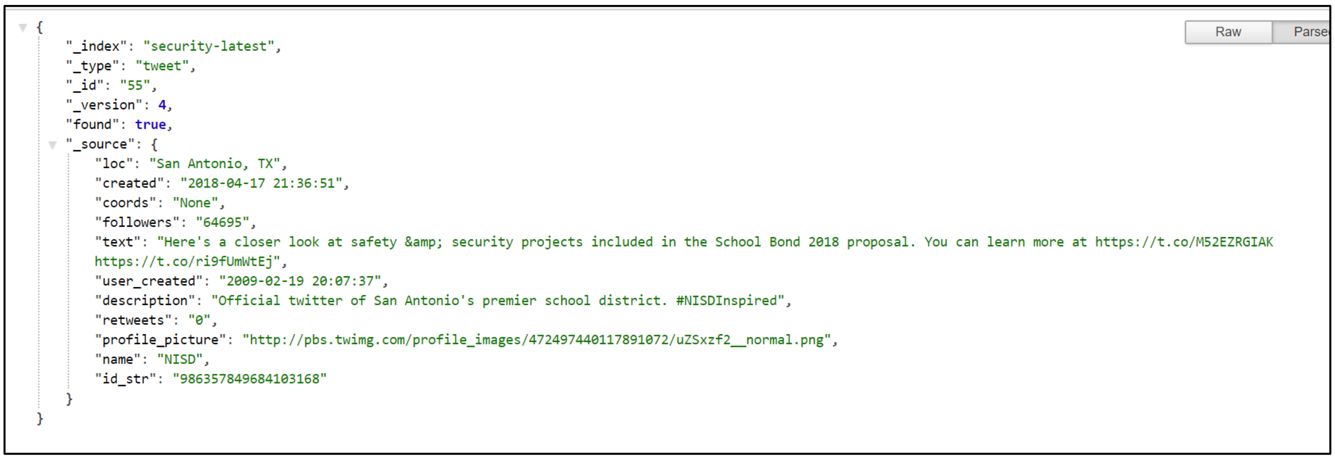
```
"total": 62554,
"hour_breakdown": {
    "0": 1881,
    "1": 1859,
    "2": 1845,
    "3": 1694,
    "4": 1613,
    "5": 1632,
    "6": 1809,
```

Figure 22. The rest of the above figure 21. This is the rest of the entries breakdown which includes the total collected tweets and the hour breakdown.

The reason that metadata is being stored rather than the actual tweets is because it would take massive storage capacities. An example of this would be in figure 22, it shows 62554 tweets for a single day and tweets are collected over a 30-day period so the total number of results for storing tweets would be 1876620 instead of just 30. It's important to note that this is just for one topic too, and this would be repeated for every one of them. The reason that this metadata is collected based on the last 30 days is because it makes it possible to detect trends over times based on patterns that occur in the data and is needed for a statistical approach to trend detection. It is used to populate the '*topic-median*' index.

3.2.2.2 What the 'topic-latest index' stores

The '*topic-latest*' index is for storing the tweets for the current date in real time. This index stores the tweets and is used in combination with the topic-median to detect real time trends. The information that is stored in an entry is: the location, timestamp of when the tweet was made, the coordinates of the tweet (This is normally blank and not used, but its included in my Tweet object), the number of followers, the tweets text, the signup date, the users description/bio for their profile, the number of retweets, their profile picture (link), their username, and the tweets id.



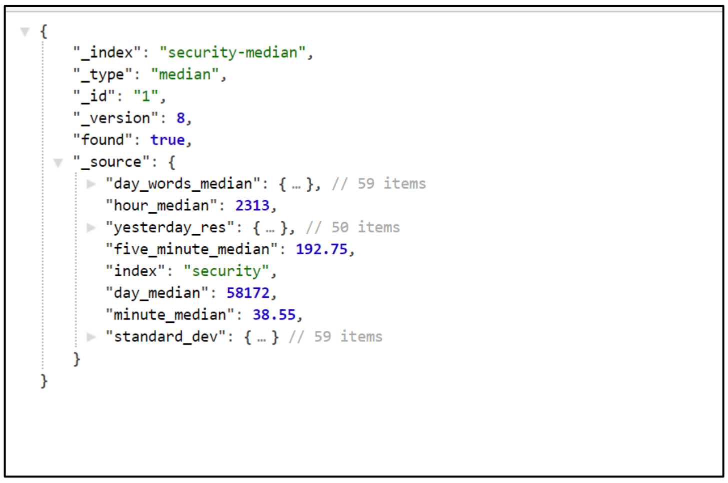
```
{
    "_index": "security-latest",
    "_type": "tweet",
    "_id": "55",
    "_version": 4,
    "found": true,
    "_source": {
        "loc": "San Antonio, TX",
        "created": "2018-04-17 21:36:51",
        "coords": "None",
        "followers": "64695",
        "text": "Here's a closer look at safety &amp; security projects included in the School Bond 2018 proposal. You can learn more at https://t.co/M52EZRGIAK
        https://t.co/ri9fUmWtEj",
        "user_created": "2009-02-19 20:07:37",
        "description": "Official twitter of San Antonio's premier school district. #NISDInspired",
        "retweets": "0",
        "profile_picture": "http://pbs.twimg.com/profile_images/472497440117891072/uZSxzf2__normal.png",
        "name": "NISD",
        "id_str": "986357849684103168"
    }
}
```

Figure 23. An example of the topic-latest index with all the fields included.

From this index, the text is mostly what is being used. The entries will be taken for the last 5 minutes which makes the '*created*' field important. This is to detect real time trends which is discussed in *Section 5*. This index is cleared every night at midnight and aggregated as metadata to the '*topic*' index.

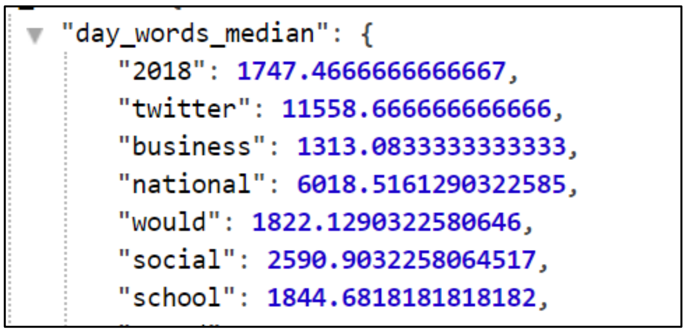
3.2.2.3 What the 'topic-median index' stores

The '*topic-median*' index will give statistical information for the index that will be used in the trend detection algorithm. It will only ever contain at most one entry in the database. This entry however will be changed based on the values for the last 30 days that are contained in the '*topic*' index. The statistical information that the *'topic-median'* has is: Standard Deviation from the mean for each of the common words found in the '*topic*' index, the median for each of the common words based on the '*topic*' index, an hour median calculated from the hour breakdowns found in the entries of the '*topic*' index, a day breakdown based on the '*topic*' index, a minute median based as an estimate of the hour breakdown of each day and divided by 60, a 5 minute median based on the 1 minute median, and also a yesterdays results field which contains a breakdown of common words yesterday which is useful tracking over time.



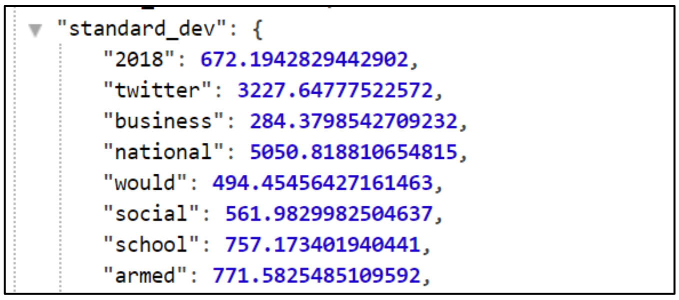
```
{
    "_index": "security-median",
    "_type": "median",
    "_id": "1",
    "_version": 8,
    "found": true,
    "_source": {
        "day_words_median": { … }, // 59 items
        "hour_median": 2313,
        "yesterday_res": { … }, // 50 items
        "five_minute_median": 192.75,
        "index": "security",
        "day_median": 58172,
        "minute_median": 38.55,
        "standard_dev": { … } // 59 items
    }
}
```

Figure 24. An example of the 'median-index'. This figure shows the hour_median, five_minute_median, minute_median, and day_median.



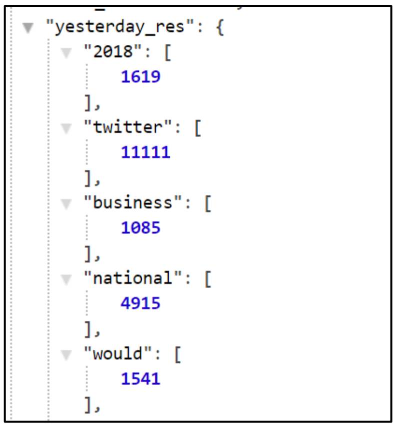
```
"day_words_median": {
    "2018": 1747.4666666666667,
    "twitter": 11558.666666666666,
    "business": 1313.0833333333333,
    "national": 6018.5161290322585,
    "would": 1822.1290322580646,
    "social": 2590.9032258064517,
    "school": 1844.6818181818182,
```

Figure 25. An example of the day_words_median field.



```
"standard_dev": {
    "2018": 672.1942829442902,
    "twitter": 3227.64777522572,
    "business": 284.3798542709232,
    "national": 5050.818810654815,
    "would": 494.45456427161463,
    "social": 561.9829982504637,
    "school": 757.173401940441,
    "armed": 771.5825485109592,
```

Figure 26. An example of the standard_dev field.

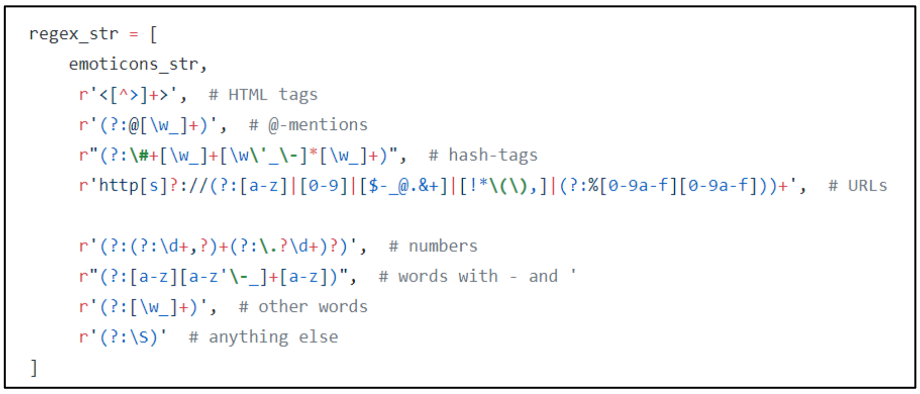


```
"yesterday_res": {
    "2018": [
        1619
    ],
    "twitter": [
        11111
    ],
    "business": [
        1085
    ],
    "national": [
        4915
    ],
    "would": [
        1541
    ],
```

Figure 27. An example of the yesterday_res field.

4. Pre-processing the Data

When the data is gathered for analysis, many actions need to be carried out before it can be analysed properly. This happens at separate points in the project. For pre-processing, a utility class was made and uses features from the NLTK (Natural Language Tool Kit) library [8]. One of the first things that is done is stripping the Tweets using regular expressions (This will allow tokenization which will be discussed in the next section).



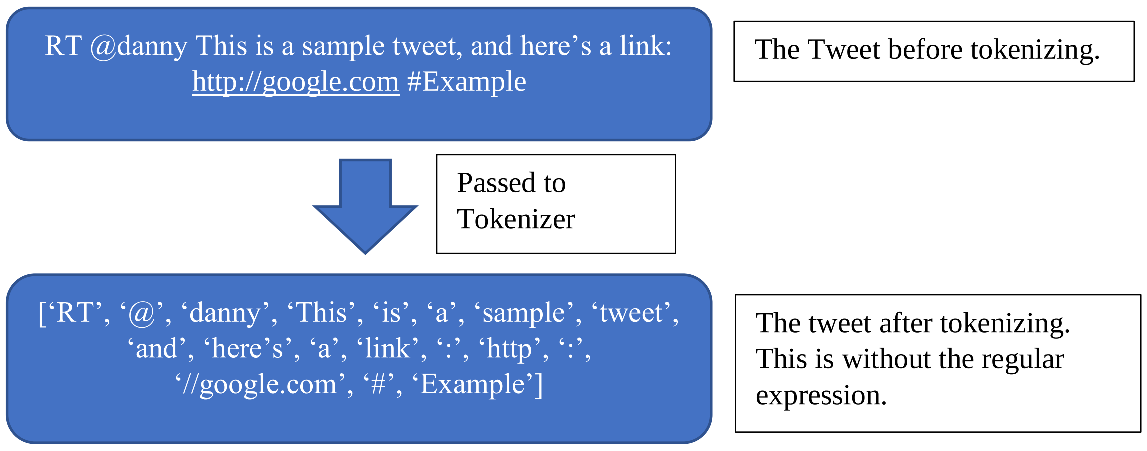
```
regex_str = [
    emoticons_str,
     r'<[^>]+>',  # HTML tags
     r'(?:@[\w_]+)',  # @-mentions
     r"(?:\#+[\w_]+[\w\'_\-]*[\w_]+)",  # hash-tags
     r'http[s]?://(?:[a-z]|[0-9]|[$-_@.&+]|[!*\(\),]|(?:%[0-9a-f][0-9a-f]))+',  # URLs

     r'(?:(?:\d+,?)+(?:\.?\d+)?)',  # numbers
     r"(?:[a-z][a-z'\-_]+[a-z])",  # words with - and '
     r'(?:[\w_]+)',  # other words
     r'(?:\S)'  # anything else
]
```

Figure 28. An example of the regex that is used.

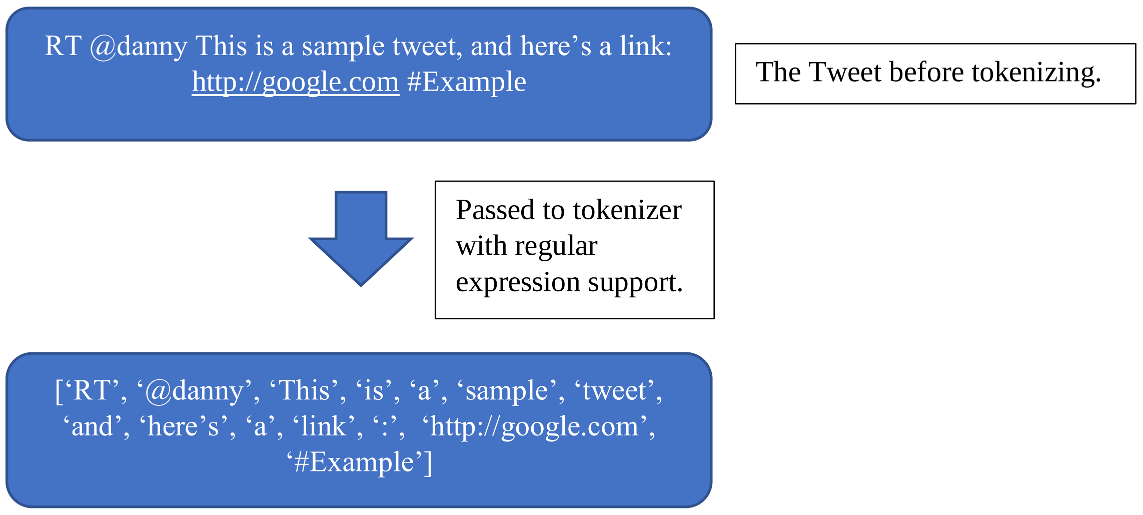
4.1 Tokenize the Tweet

The first step in pre-processing any Tweet is to tokenize it. The way a Tweet comes back from Twitter to the project is in String format just like a sentence would. However, it's not useful to analyse Tweets based on a sentence by sentence comparison. It is much more useful to compare Tweets via individual words e.g. finding common words between them. This is what tokenizing does. It will break down the Tweet into individual words that can be then analysed. As discussed in Section 4 "*Pre-processing the Data*" a regular expression will be used to help with this.

Example without regular expression:


RT @danny This is a sample tweet, and here’s a link: http://google.com #Example
The Tweet before tokenizing.
Passed to Tokenizer
[‘RT’, ‘@’, ‘danny’, ‘This’, ‘is’, ‘a’, ‘sample’, ‘tweet’, ‘and’, ‘here’s’, ‘a’, ‘link’, ‘:’, ‘http’, ‘:’, ‘//google.com’, ‘#’, ‘Example’]
The tweet after tokenizing. This is without the regular expression.

Example with regular expression:

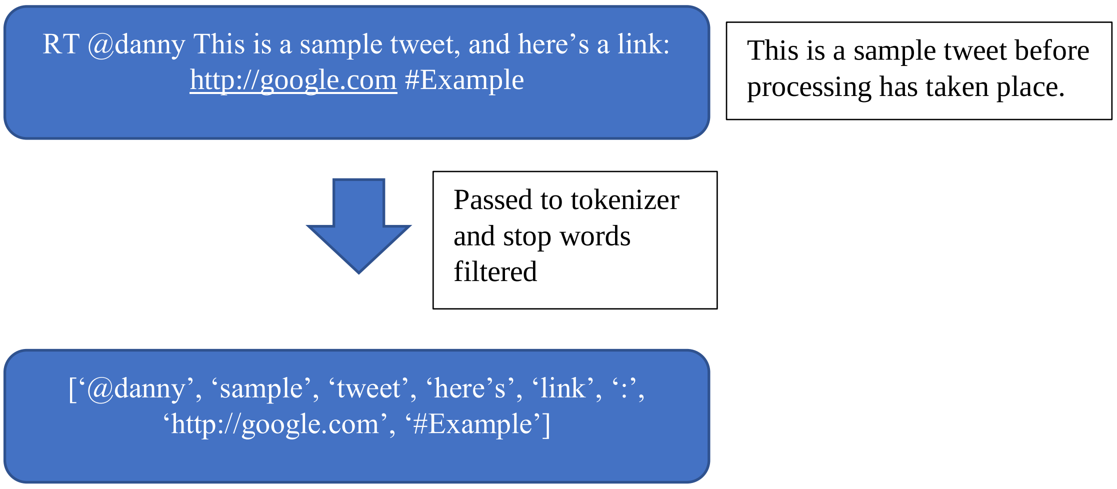

RT @danny This is a sample tweet, and here’s a link: http://google.com #Example
The Tweet before tokenizing.
Passed to tokenizer with regular expression support.
[‘RT’, ‘@danny’, ‘This’, ‘is’, ‘a’, ‘sample’, ‘tweet’, ‘and’, ‘here’s’, ‘a’, ‘link’, ‘:’, ‘http://google.com’, ‘#Example’]

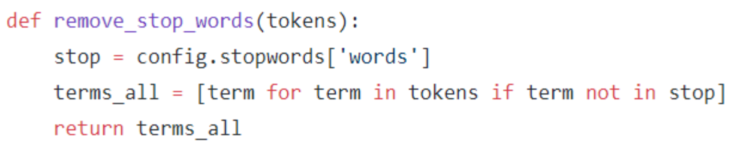
In this project, the regular expression example will always be used when pre-processing a Tweet.

4.2 Stop words

Another issue that can be solved using Natural Language Processing(NLP) and the NLTK is the issue that stop words cause. Stop words are usually words that are extremely common in a language. These words need to be filtered out as it would skew analysis since they are so common. An example of some stop words would be: a, it, the, and I. The NLTK package comes with its own built in stop words list for the English language and this is what the stop words list in this project is based off. However, Twitter has its own stop words that are only relevant to Twitter. This includes words like ‘rt’ (Retweet), and ‘dm’ (Direct message, or private message) that would also need to be filtered out. The NLTK set was more of a starting point and has been expanded on over time. With this project, a separate stop words list can be supplied in the configuration file which makes it flexible. This is done every time Tweets are processed also because if this wasn’t done, all the trends that are found would just be words like ‘the’.

Example of removing Stop Words:


RT @danny This is a sample tweet, and here’s a link: http://google.com #Example
This is a sample tweet before processing has taken place.
Passed to tokenizer and stop words filtered
[‘@danny’, ‘sample’, ‘tweet’, ‘here’s’, ‘link’, ‘:’, ‘http://google.com’, ‘#Example’]



```
def remove_stop_words(tokens):
    stop = config.stopwords['words']
    terms_all = [term for term in tokens if term not in stop]
    return terms_all
```

Figure 29. A sample of the function that removes stopwords.

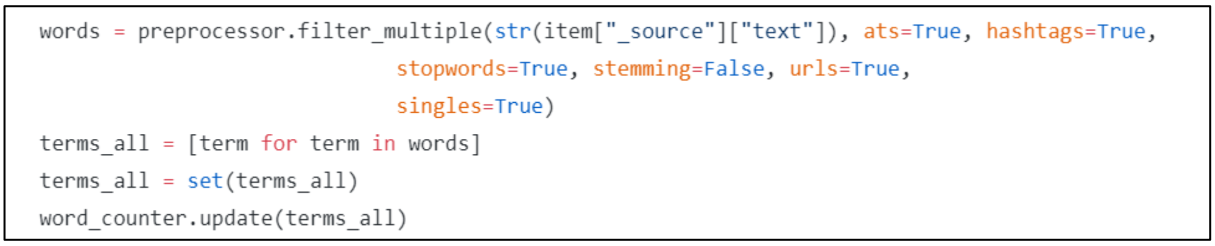
4.3 Other processing techniques

The project also uses some other processing techniques in certain areas and unlike the 2 discussed above in section 4.1 and section 4.2 are not used every time a Tweet is processed. This includes:

1. Remove mentions/@'s: This processing technique will remove usernames from a Tweet so anywhere in a Tweet where an at sign is placed, it is removed e.g. *'@danny this is a tweet'* would become '*this is a tweet*'. Most of the time when storing common words this will be used since the 'at' has no real meaning towards detecting a pattern. This is used when aggregating to the '*topic*' instance.
2. Remove hashtags: This processing technique will remove hashtags from a Tweet. This function isn't really that useful, but a lot of the time with processing the message is got from outside of the hashtag.
3. Remove single characters: This processing technique will remove single characters from the Tweet. Sometimes characters can bypass the stop words list such as a character like '^' and this shouldn't be tracked in a common words list. This is used when aggregating to the '*topic*' index.
4. Remove urls: This processing technique will remove any links from the Tweet. This is good for avoiding spam. It is used when aggregating to the '*topic*' index.
5. Stemming: This processing technique is done using Porters Stemming Algorithm which is provided by NLTK. It basically makes similar words into one e.g. hacked becomes hack. The problem with this is that it also truncates a lot of useful information such as 'Google' becoming 'Googl'.

4.4 How does this fit into the project

Pre-processing is done in a few different areas on the project. One of the first is on the Celery Task (Discussed in Section 6) that is detecting real time trends. It does this to avoid the pointless words, and it gives it the ability to check the values vs the ones that are maintained on the '*median*' index. It uses everything except the stemming function to do this.



```
words = preprocessor.filter_multiple(str(item["_source"]["text"]), ats=True, hashtags=True,
                                     stopwords=True, stemming=False, urls=True,
                                     singles=True)
terms_all = [term for term in words]
terms_all = set(terms_all)
word_counter.update(terms_all)
```

Figure 30. An example of it being used for real time trend detection. It updates a counter to return how many times a word has appeared.

There's a task that is run at midnight that purges the '*latest*' index and it also uses the same pre-processing technique as figure 30. This is to count the number of occurrences of certain words for that day, and then store the most common ones in the *'median'* index.

It's also used in the collecting of old tweets, and the suggest topic functions to avoid given keywords that are stop words or not relevant. Without pre-processing, trend detection would not be possible.

5. Real Time Detection – Check against detection times

Real time detection as part of this project is all about trying to detect trends quickly. This has several different approaches in practice with Twitter not disclosing publicly their algorithm for how they do it. For trend detection, many different approaches exist, and some have been analysed for their use in this project. One approach that was implemented in an earlier version of this project was using Machine Learning and will be discussed in the reflections section. Something that is implemented in this project in its simplest sense is a spam detection system. This checks for duplicate tweets, users and only counts them as one value.

5.1 The Trend Detection Algorithm

The approach I ended up going with was a combination of a few different statistical approaches. It uses the values stored in the '*median*' index for each topic to compare values against and the basic premise is that it will come up with a number which will be the likelihood of it being a trend. This algorithm uses the following fields from the '*median*' index:

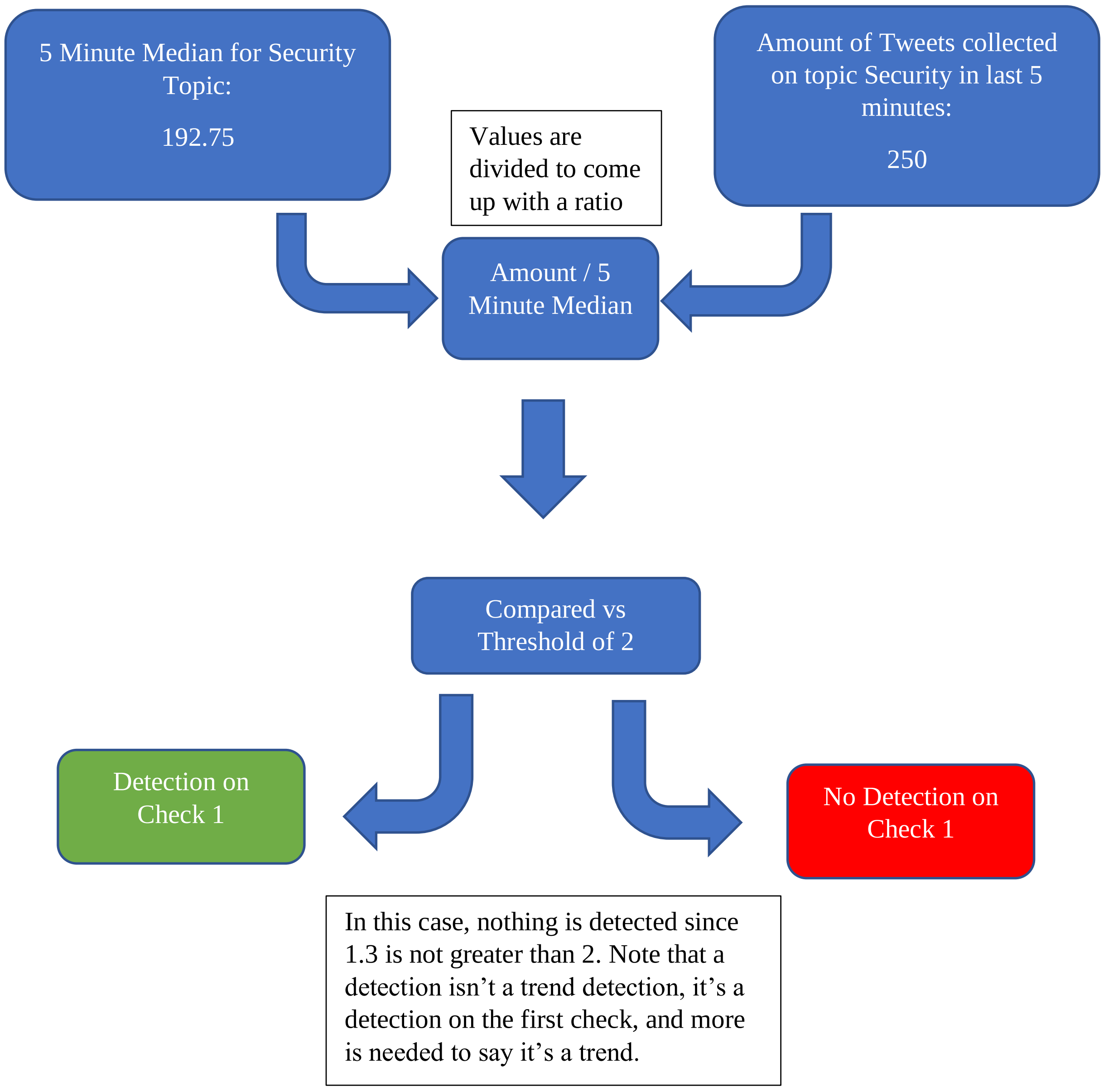
1. Five-minute median
2. Yesterday's results
3. Hour median (Used for task run hourly)

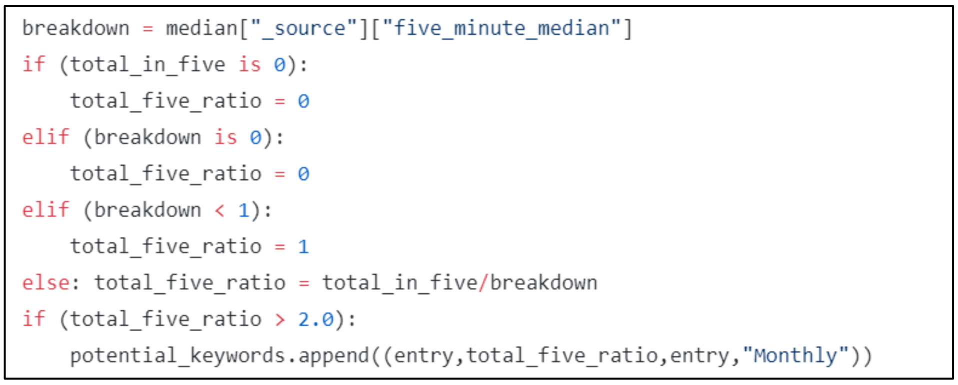
The basic process is as follows:

5.1.1 Check 1 – 5 Minute Median

The first check is to see if the topic is above the 5-minute median. This first check is just based on the topic alone (doesn't take keywords into account). This is compared against a threshold value which is calculated by dividing the current rate for the last 5 minutes vs the median value.

Example:


5 Minute Median for Security Topic:
192.75
Amount of Tweets collected on topic Security in last 5 minutes:
250
Values are divided to come up with a ratio
Amount / 5 Minute Median
Compared vs Threshold of 2
Detection on Check 1
No Detection on Check 1
In this case, nothing is detected since 1.3 is not greater than 2. Note that a detection isn't a trend detection, it's a detection on the first check, and more is needed to say it's a trend.



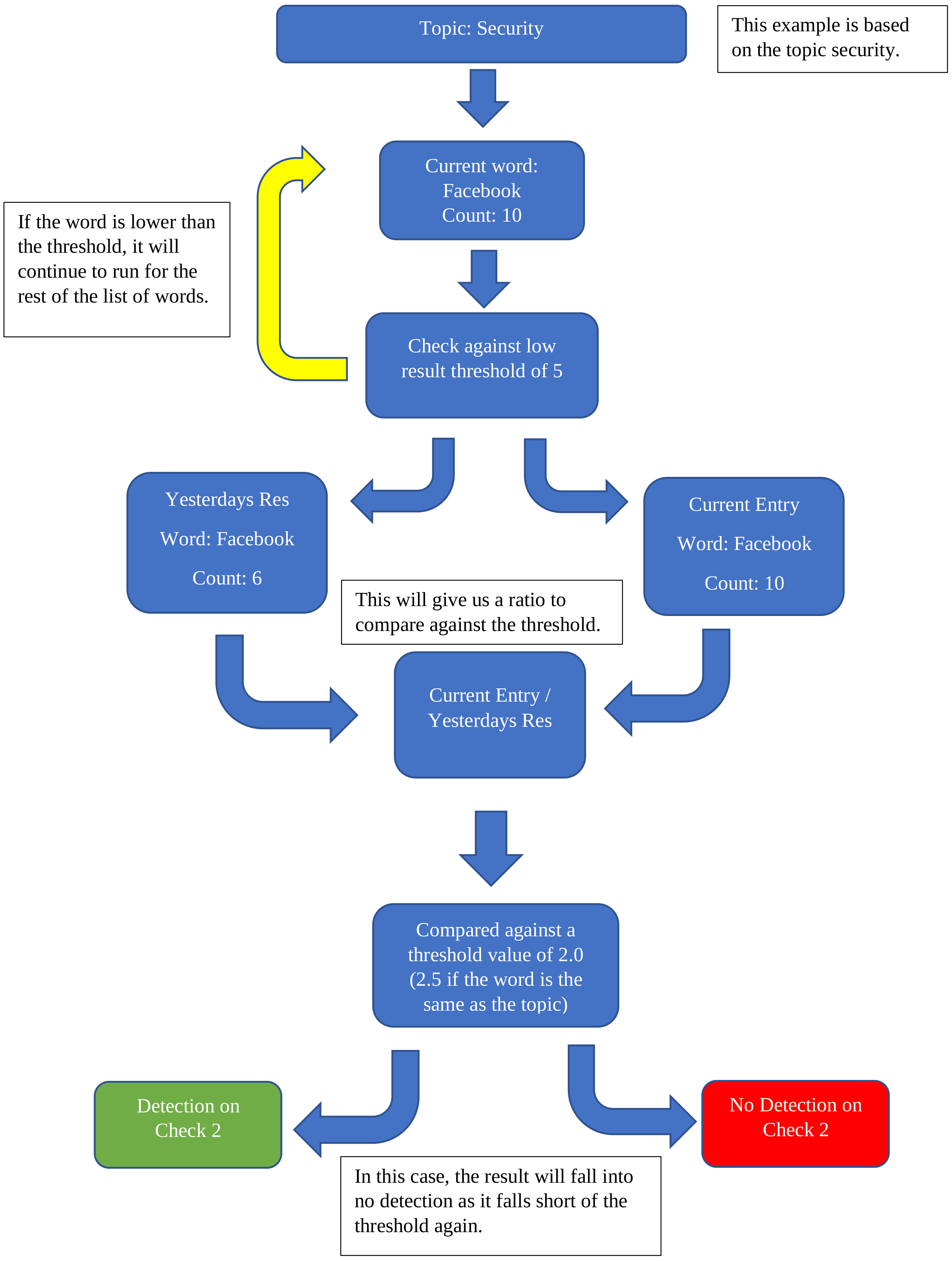
```
breakdown = median["_source"]["five_minute_median"]
if (total_in_five is 0):
    total_five_ratio = 0
elif (breakdown is 0):
    total_five_ratio = 0
elif (breakdown < 1):
    total_five_ratio = 1
else: total_five_ratio = total_in_five/breakdown
if (total_five_ratio > 2.0):
    potential_keywords.append((entry,total_five_ratio,entry,"Monthly"))
```

Figure 31. An example of the code that makes this happen. total_in_five is a variable containing the number of tweets for the last 5 minutes and is divided by the breakdown which is the value of the median index. This is checked vs a threshold of 2.

5.1.2 Check 2 – Words checked against yesterday

The next check that is done is to check if the words appeared in the index yesterday. This is to pick up trends that are continuing to rise in popularity over time and append them to existing entries if they already exist. It goes through every word that appears for the last 5 minutes of a topic that has been tokenized and pre-processed as shown in Section 4. There is one issue with doing this, and that is just because a word wasn't tweeted yesterday doesn't mean it's a trend and the word could be tweeted just once. To prevent this a check is in place to say that the current word needs to appear more than 5 times in the current 5-minute section. It is then compared in a similar way to the previous example. If the word isn't in either the words from yesterday or all time, another check will be responsible for it:

Example when word is in yesterday's results:



Topic: Security
This example is based on the topic security.
Current word: Facebook Count: 10
If the word is lower than the threshold, it will continue to run for the rest of the list of words.
Check against low result threshold of 5
Yesterdays Res Word: Facebook Count: 6
Current Entry Word: Facebook Count: 10
This will give us a ratio to compare against the threshold.
Current Entry / Yesterdays Res
Compared against a threshold value of 2.0 (2.5 if the word is the same as the topic)
Detection on Check 2
No Detection on Check 2
In this case, the result will fall into no detection as it falls short of the threshold again.

5.1.3 Check 3 – Against common words for the past 30 days

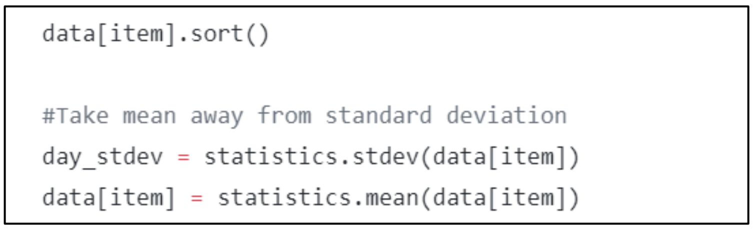
The next check that the algorithm does is a check against the words stored in the ‘*median*’ index. This does not include the words collected for yesterday, but rather the median that has been collected for the common words based on the past 30 days. It works much like Check 1 and Check 2 with the difference being that this check is more important than the last 2 and is more heavily weighted at the end. This check uses both the Median values and standard deviation from the mean also. It is first important to understand how the Standard deviation is being calculated because it involves more logic than the previous examples. In the ‘*median*’ index the mean of the common words isn’t being stored. However, when this task is run the mean is also calculated. Before going through exactly how this works, the way the median is stored should be addressed.

Median

The median is calculated based on the values over 30 days. The method to calculate this works a bit different than just simply getting the mean of 30 values since there can be many edge cases. The solution to this was to make it so that words that appear 10 times or greater are calculated to have a Median. The original way it was done in an early version of the project was to add 0’s until that keyword had 30 entries, but this lead to lots of common words for an index having the value of 0. An example of how it would work in the current version would: Given an array of [1,2,3,4,5,6] the median would be 4.

Standard Deviation from the Mean

When a task is run, the mean is calculated along with the Median but not stored. Using the list of values, it is simple to calculate the standard deviation using the Python statistics package. The standard deviation for the word is then stored along with the Median value to the ‘*median*’ index.



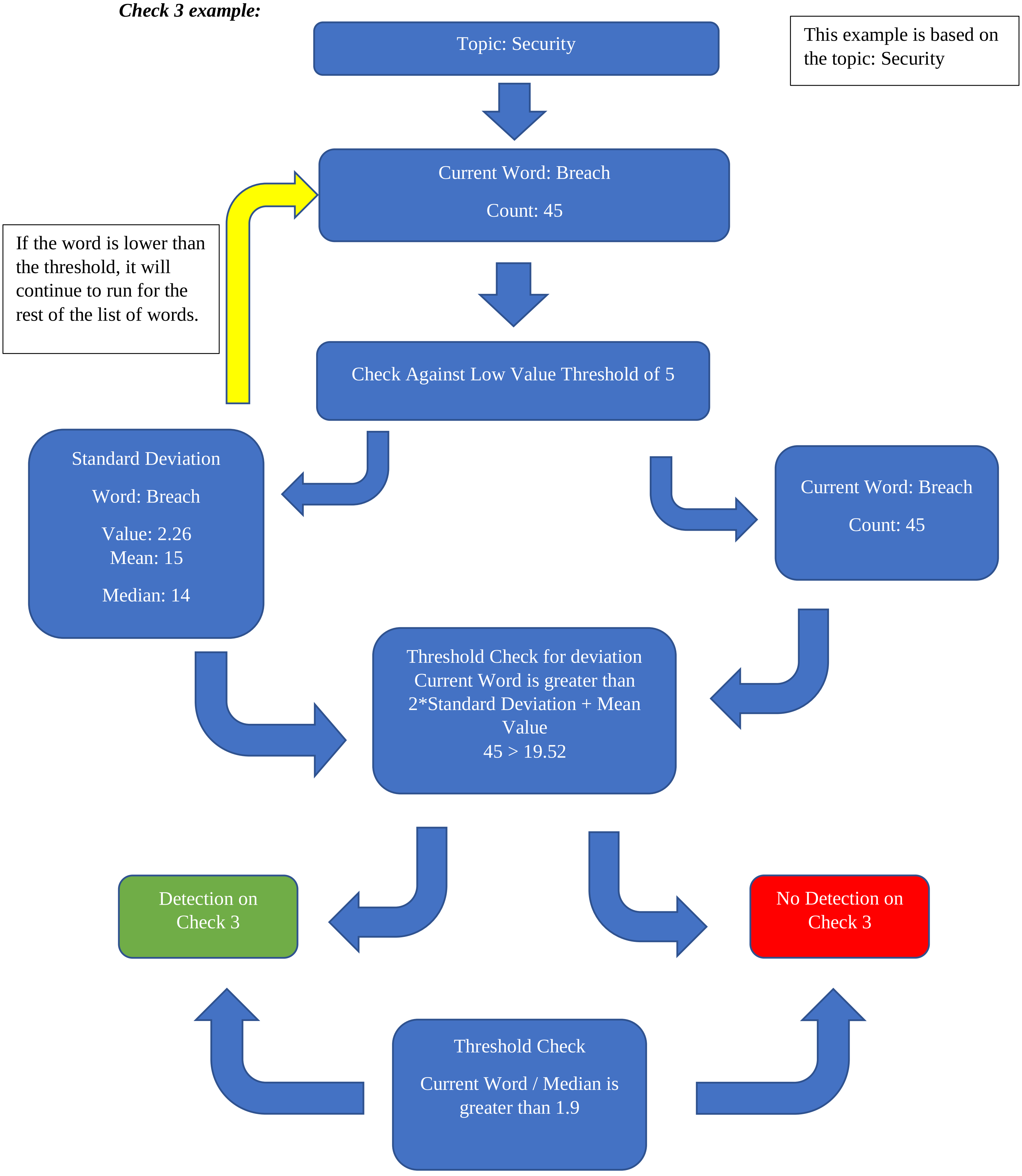
```
data[item].sort()

#Take mean away from standard deviation
day_stdev = statistics.stdev(data[item])
data[item] = statistics.mean(data[item])
```

Figure 33. An example of how the standard deviation is being calculated.

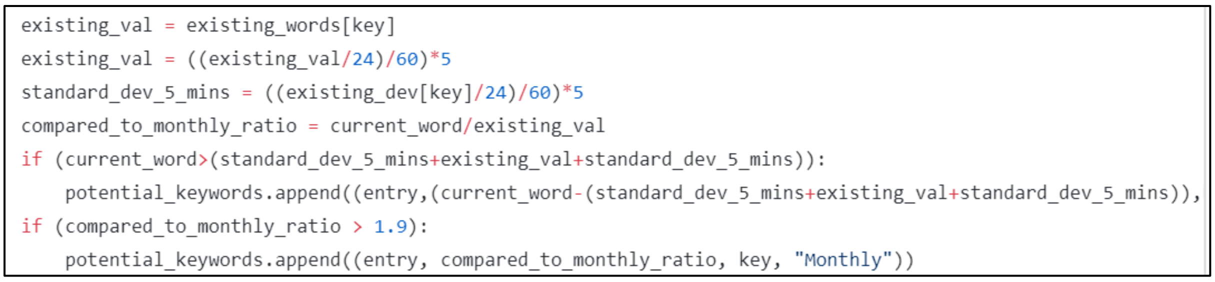
Check 3 example:

This example is based on the topic: Security

If the word is lower than the threshold, it will continue to run for the rest of the list of words.



In this example, both threshold checks correspond to detections. The standard deviation check means that it is over 2 standard deviations away from the mean value which is a value worked out over time, and the threshold check in this case is set to 1.9 for better performance. If one passed and the other didn't, it will still be marked as a detection, but with less likelihood.



```
existing_val = existing_words[key]
existing_val = ((existing_val/24)/60)*5
standard_dev_5_mins = ((existing_dev[key]/24)/60)*5
compared_to_monthly_ratio = current_word/existing_val
if (current_word>(standard_dev_5_mins+existing_val+standard_dev_5_mins)):
    potential_keywords.append((entry,(current_word-(standard_dev_5_mins+existing_val+standard_dev_5_mins)),
if (compared_to_monthly_ratio > 1.9):
    potential_keywords.append((entry, compared_to_monthly_ratio, key, "Monthly"))
```

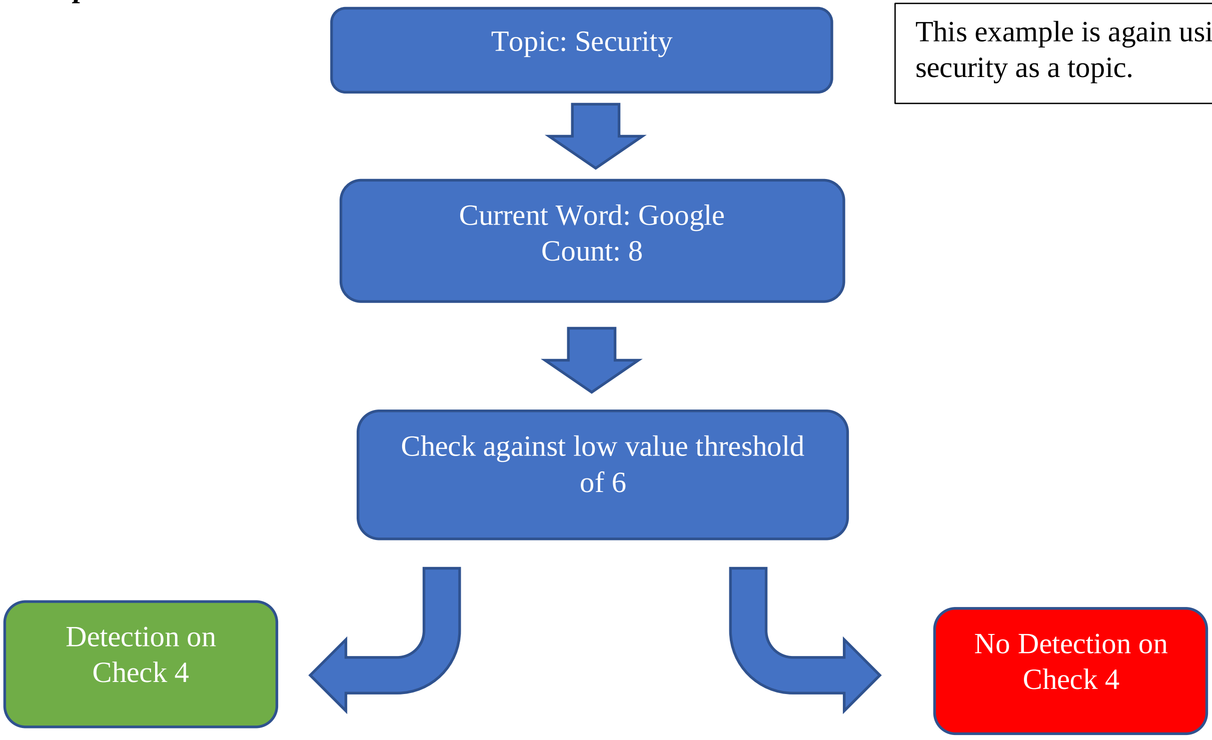
Figure 34. The code responsible for both detecting on standard deviations and the ratio threshold.

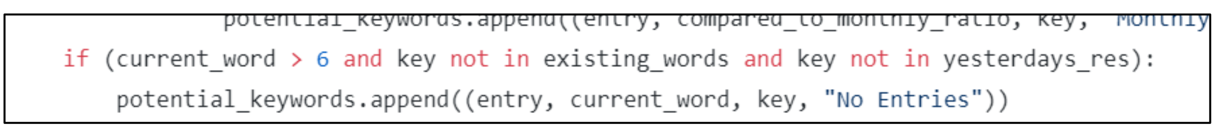
5.1.4 Check 4 – Unknown words

The last check is for unknown words. This is for words that have no previous metadata stored about them and the check is relatively simple. It does a minor check against the word count for this word to ensure that it hasn't just been tweeted about once (bear in mind that a word that appears in a Tweet more than once will still appear as 1.).

Example Check 4

This example is again using security as a topic.



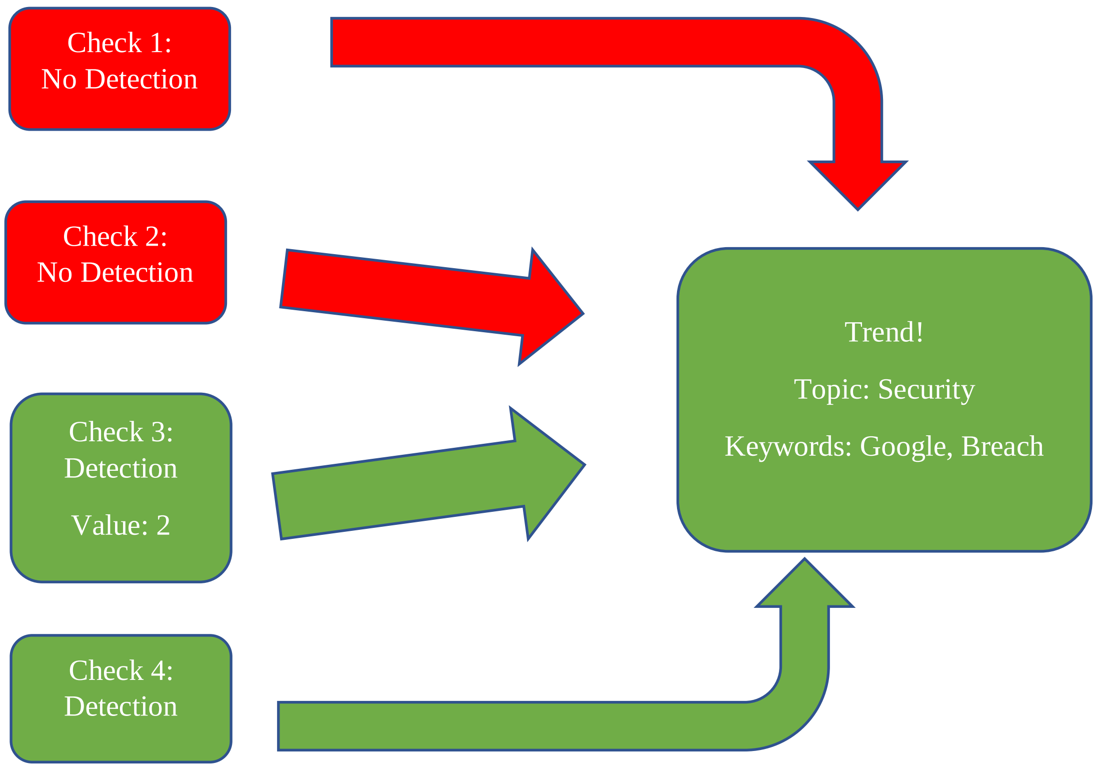


```
        potential_keywords.append((entry, compared_to_monthly_ratio, key, "Monthly"
    if (current_word > 6 and key not in existing_words and key not in yesterdays_res):
        potential_keywords.append((entry, current_word, key, "No Entries"))
```

Figure 35. The code that is responsible for this check.

5.1.5 Piecing the checks together to detect a trend

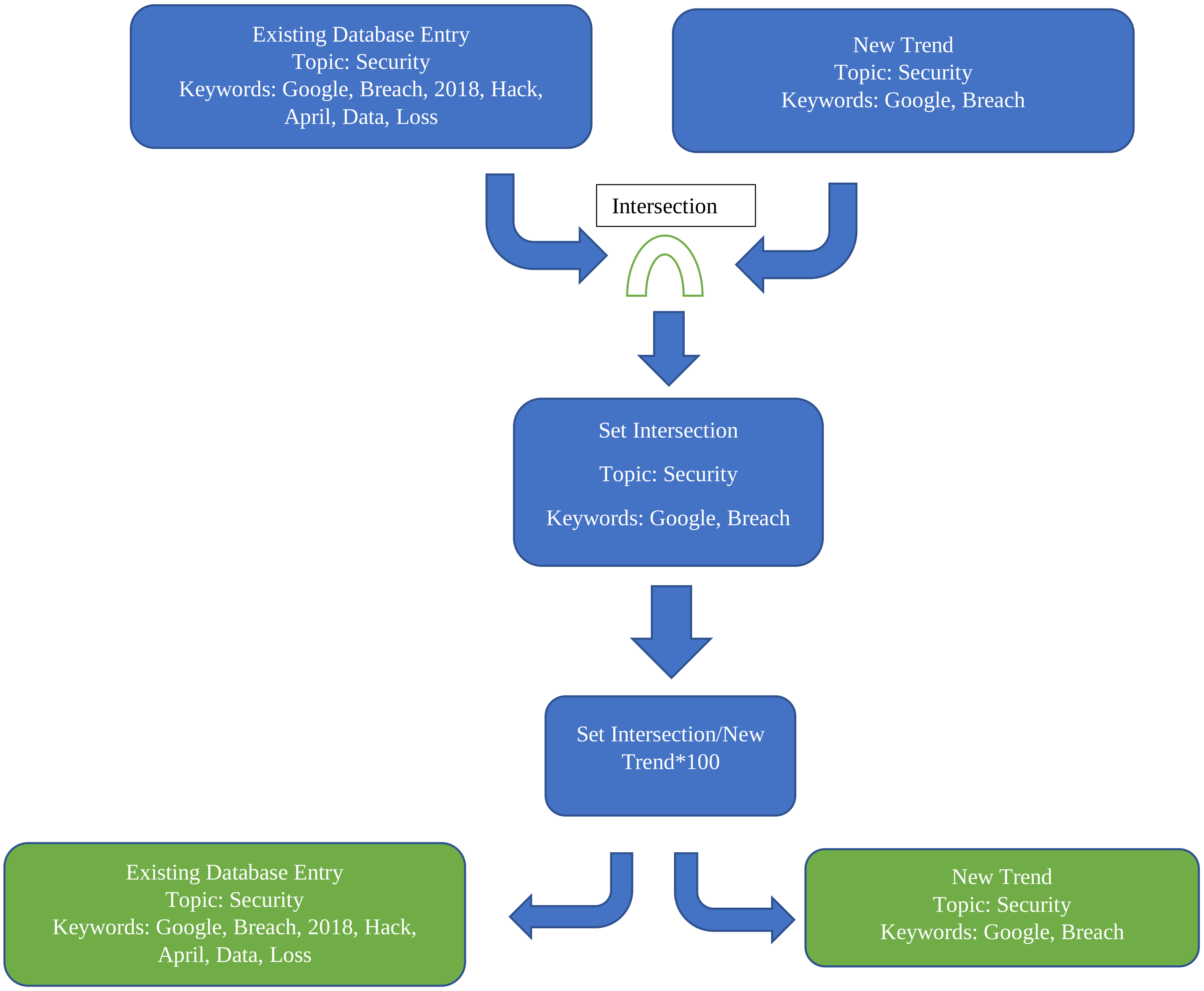
The last part of running the algorithm is straight forward in the sense that all the work is done and now it is only a matter of piecing together the results. This works by checking which of the checks has reported that it is a detection on that check. It then compares this amount to a threshold to tell which trends are. The threshold value is 0.6 or 3/5 for this. It is important to note that check 3 is double weighted if both values are trends. So, for the example done for the 4 checks in this example:



A trend has been detected and in this case it's easy to piece together that the story has something to do with a Google security breach.

5.2 Appending to an existing trend or a new trend

Now the system must decide what to do with this trend. The process that it uses for this is set theory. It first makes the keywords a set (Now a keyword can only appear once), and if there's no other entry for that topic it will save it. However, if there's an existing entry for that topic in the Database, it will check the new trends keywords vs the existing entry in the database. If this new entry intersected with the database entry is 60% the same as the new set, then the entry is appended. For example:



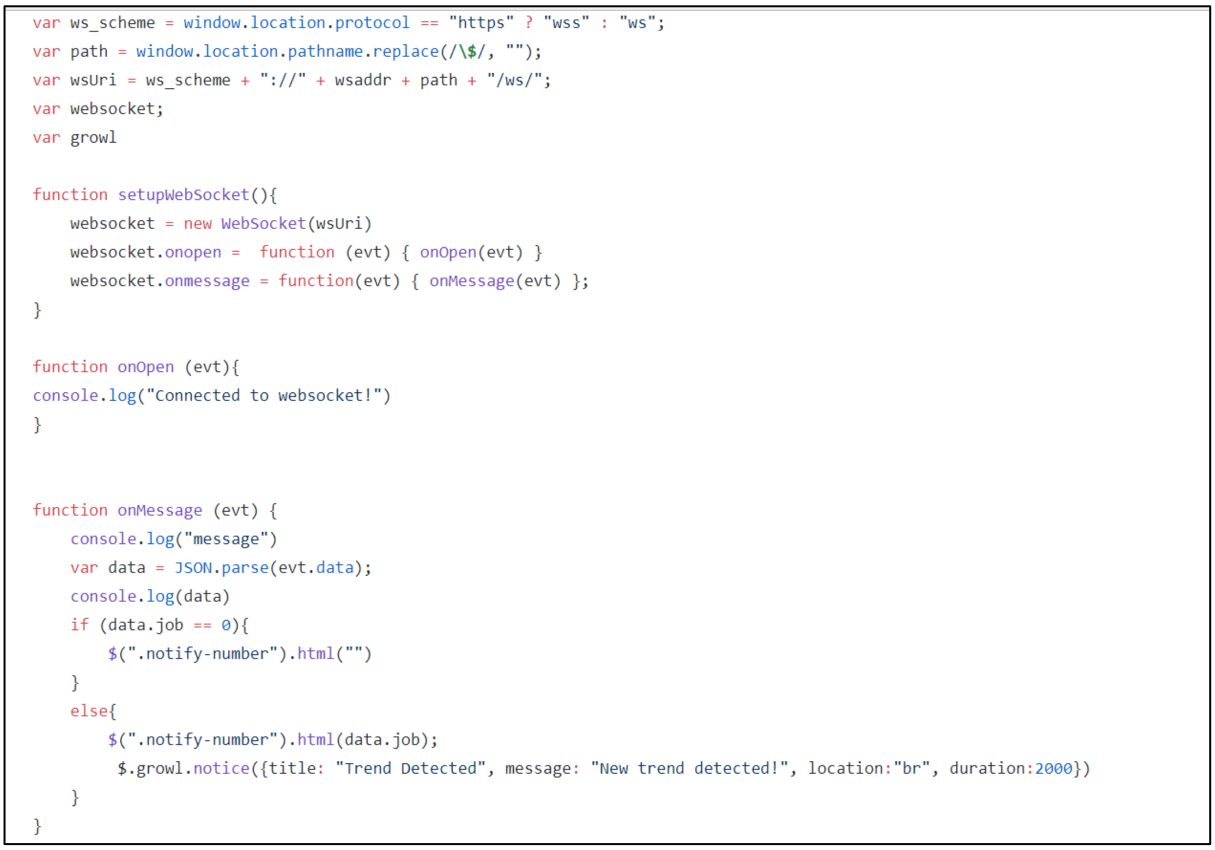
In this case, since the outcome of ‘Set Intersection / New Trend *100’ is greater than 60%, it would get added to the existing entry.

6. Asynchronous Channels and Celery

6.1 Django Channels

Django Channels is a new technology for the Django web framework which is to be included by default in the newer versions [9]. By default, Django only supports HTTP which means that it cannot use web sockets, iot protocols, and chat protocols. For this project this is a big deal because the web sockets are extremely useful for given live updates to the user. Django Channels is a solution to this problem. It does this by adding an additional ‘*asynchronous*’ layer to Django which allows Django to be both asynchronous, and synchronous [9].

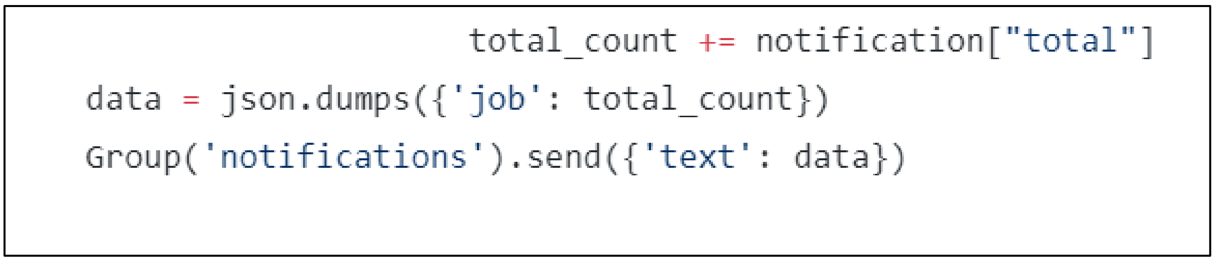
In this project, Django Channels is needed to notify the user in real time on any page they visit. If this wasn’t done asynchronously through websockets it would mean that the user would not be able to notify a user on every page (This is because page loads in Django are synchronous). Using this technology, a websocket is opened on every web page that is loaded and it will act as a listener for when a notification comes in.



```
var ws_scheme = window.location.protocol == "https" ? "wss" : "ws";
var path = window.location.pathname.replace(/\$/, "");
var wsUri = ws_scheme + "://" + wsaddr + path + "/ws/";
var websocket;
var growl

function setupWebSocket(){
    websocket = new WebSocket(wsUri)
    websocket.onopen =  function (evt) { onOpen(evt) }
    websocket.onmessage = function(evt) { onMessage(evt) };
}

function onOpen (evt){
console.log("Connected to websocket!")
}


function onMessage (evt) {
    console.log("message")
    var data = JSON.parse(evt.data);
    console.log(data)
    if (data.job == 0){
        $(".notify-number").html("")
    }
    else{
        $(".notify-number").html(data.job);
         $.growl.notice({title: "Trend Detected", message: "New trend detected!", location:"br", duration:2000})
    }
}
```

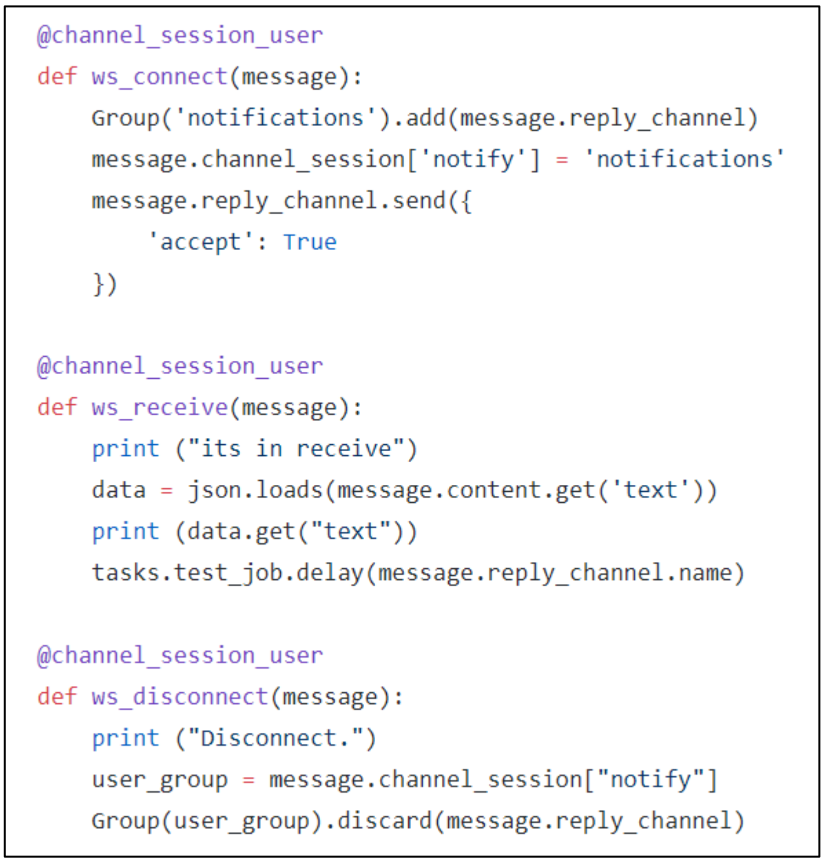
Figure 36. An example of the JavaScript that is run on webpage load.

The way that this code works on the JavaScript side is straightforward. The function onOpen() runs when the page is loaded. The variable for ‘*wsUri*’ works by concatenating the protocol (HTTP/HTTPS) with the current web address. The function onMessage() appends the notification bell on the front end, and also the toast notification at the bottom using the growl notification.



```
                    total_count += notification["total"]
    data = json.dumps({'job': total_count})
    Group('notifications').send({'text': data})
```

Figure 36. An example of sending the notification from the Celery task using Django Channels. This sends the data to the Group 'notifications' code which will be shown below.



```
@channel_session_user
def ws_connect(message):
    Group('notifications').add(message.reply_channel)
    message.channel_session['notify'] = 'notifications'
    message.reply_channel.send({
        'accept': True
    })

@channel_session_user
def ws_receive(message):
    print ("its in receive")
    data = json.loads(message.content.get('text'))
    print (data.get("text"))
    tasks.test_job.delay(message.reply_channel.name)

@channel_session_user
def ws_disconnect(message):
    print ("Disconnect.")
    user_group = message.channel_session["notify"]
    Group(user_group).discard(message.reply_channel)
```

Figure 37. This is the backend logic that makes it possible to send messages to the group and accessed on the front-end JavaScript.

The ws_connect, and the ws_disconnect functions are the main ones that make asynchronous Django possible. The connect function sets up a group called '*notifications*' which is used to send the data shown in Figure 36. On the disconnect function, the group is deleted.

6.2 Celery Tasks

Celery is one of the main driving forces behind this project. Celery is an asynchronous task queue that can run scheduled tasks along with real-time tasks. It does so in an asynchronous way and uses threading to increase the performance of these queues [10].

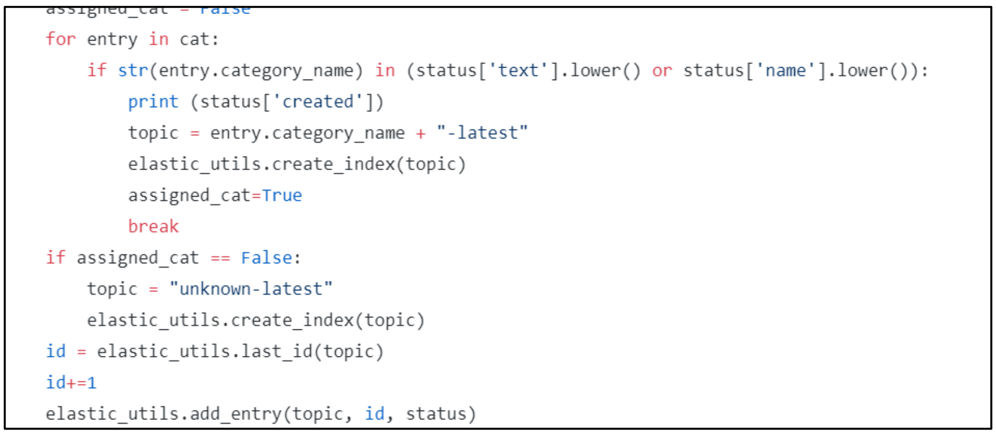
The way the queuing system works is that when a '*celery task*' is ran, it is added to the queue for that specific worker. A worker is an instance of Celery that carries out these tasks, and as many of them as you like can be set. In the case of this project 4 workers are set: Default, misc, priority_high, and old_tweets. This means that they will each have a different queue assigned to them and will run asynchronously in the background while the system is in operation.

6.2.1 Default Queue

The default queue is one set for collecting live tweets. This is the queue that will need the most processing power because it is constantly collecting tweets from every topic. It is important that with this queue that the operation is carried out on more than one thread. This can be done by setting the – concurrency setting, and the amount will depend on the number of topics. However, the concurrency that seems to work fine with is 15 for about 15 topics.

Default Queues Tasks

Aggregate_words(): The aggregate words function is what is responsible for aggregating words to the relevant Elasticsearch index. This is the only task that the default queue is responsible for and it needs to be able to keep up with the rate of Tweets because if it does not, the Tweets will no longer be accurate and will fall behind even further over time.



```
assigned_cat = False
for entry in cat:
    if str(entry.category_name) in (status['text'].lower() or status['name'].lower()):
        print (status['created'])
        topic = entry.category_name + "-latest"
        elastic_utils.create_index(topic)
        assigned_cat=True
        break
if assigned_cat == False:
    topic = "unknown-latest"
    elastic_utils.create_index(topic)
id = elastic_utils.last_id(topic)
id+=1
elastic_utils.add_entry(topic, id, status)
```

Figure 38. Sample of the aggregate words task. It checks the Tweets username and text for the topic in question and adds it to the relevant index based on this. It uses the 'unknown-latest' index as a failsafe in case it doesn't find a category which will not happen in general.

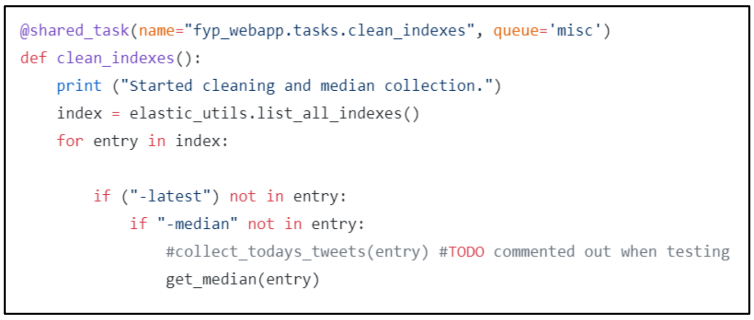
6.2.2 Misc Queue

The misc queue is responsible for tasks that are run on schedules. Celery queues can be run every so often or at certain times of the day. This will be taken advantage of in this project by setting up this queue and 2 different tasks that will be run on schedules for it:

Misc Queues Task:

*Check*_index(): This task is run every five minutes (also run on an hour check) and is what uses the algorithm discussed in section '*5.1*' to detect trends. The functionality for this queue can thus be found in section '*5.1*'.

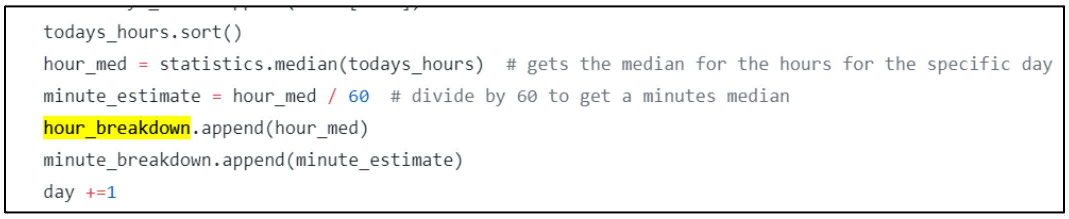
Clean_indexes(): This task is run at midnight every night and is responsible for purging the '*-latest*' index of each topic. It will then use the tweets from each date to populate the '*-median*' index for each topic with the meta data collected from each day. This function is broken into a few different sub-functions to make it cleaner so the functions that it uses are: get_median(entry), collect_todays_tweets(entry), and clean_indexes().



```
@shared_task(name="fyp_webapp.tasks.clean_indexes", queue='misc')
def clean_indexes():
    print ("Started cleaning and median collection.")
    index = elastic_utils.list_all_indexes()
    for entry in index:

        if ("-latest") not in entry:
            if "-median" not in entry:
                #collect_todays_tweets(entry) #TODO commented out when testing
                get_median(entry)
```

Figure 39. An example of the clean_indexes() task. It calls to collect_todays_tweets, and the get_median() functions for logic.

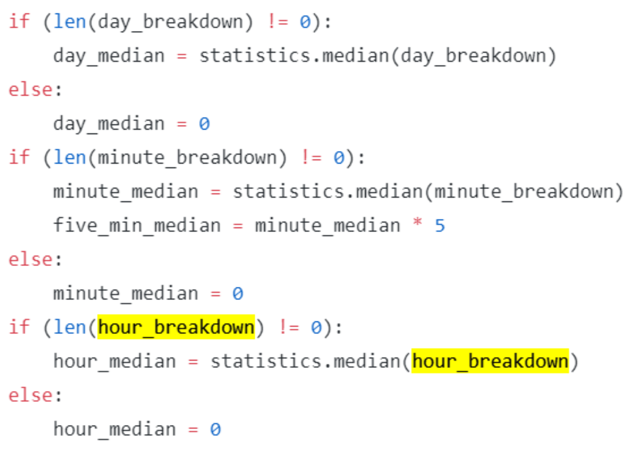
The get_median() function iterates through every entry in the '*-latest*' index and appends all the relevant information that needs to be posted to the '*median*' index. This includes the hour breakdowns:



```
todays_hours.sort()
hour_med = statistics.median(todays_hours)  # gets the median for the hours for the specific day
minute_estimate = hour_med / 60  # divide by 60 to get a minutes median
hour_breakdown.append(hour_med)
minute_breakdown.append(minute_estimate)
day +=1
```

Figure 40. Shows how the hour breakdown is applied to calculate the minute median and appending the median for a word based on the statistics package.

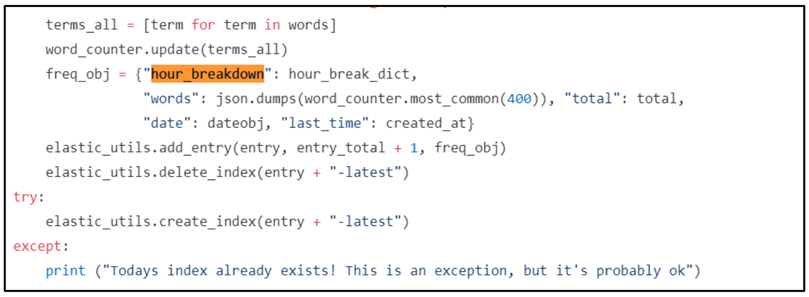
It also aggregates the most common words with a count to this entry before it is calculated alongside the median and then will change the values in accordance with the other entries in the "*topic*" index.



```
if (len(day_breakdown) != 0):
    day_median = statistics.median(day_breakdown)
else:
    day_median = 0
if (len(minute_breakdown) != 0):
    minute_median = statistics.median(minute_breakdown)
    five_min_median = minute_median * 5
else:
    minute_median = 0
if (len(hour_breakdown) != 0):
    hour_median = statistics.median(hour_breakdown)
else:
    hour_median = 0
```

Figure 41. An example of appending the new median values used for the median.

The *collect_todays_tweets()* function needs to be run before this to make sure that the new entry is in the index. This function collects all of todays tweets and passes the data to the get_median() function. Lastly, the index is then deleted and re-created with no entries in ready for the next day's results:



```
    terms_all = [term for term in words]
    word_counter.update(terms_all)
    freq_obj = {"hour_breakdown": hour_break_dict,
                "words": json.dumps(word_counter.most_common(400)), "total": total,
                "date": dateobj, "last_time": created_at}
    elastic_utils.add_entry(entry, entry_total + 1, freq_obj)
    elastic_utils.delete_index(entry + "-latest")
try:
    elastic_utils.create_index(entry + "-latest")
except:
    print ("Todays index already exists! This is an exception, but it's probably ok")
```

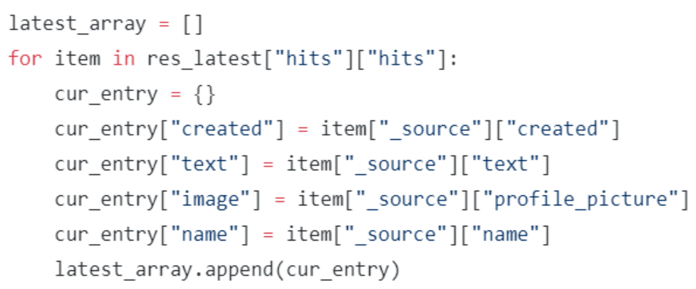
Figure 42. An example of the entry being added to the 'topic' index and then the '-latest' index being purged and recreated.

6.2.3 Priority_high Queue

The priority high queue is a queue that takes priority over the other queues. This is usually for important items on the front end to the user like loading charts or loading the web page itself. There are 2 main tasks that are using the priority high queue in this project and they both involve expensive operations to Elasticsearch.

Priority_high Queues tasks:

Elastic_stats(): This task is used in combination with loading the statistics page, it will first display a loading bar while the information is being collected asynchronously. It will query Elasticsearch for information about each index and its latest results (The 5 latest tweets for the '*topic-latest*' index, and the 10 latest days for the '*topic*' index). It will also return some Median information about each index.



```
latest_array = []
for item in res_latest["hits"]["hits"]:
    cur_entry = {}
    cur_entry["created"] = item["_source"]["created"]
    cur_entry["text"] = item["_source"]["text"]
    cur_entry["image"] = item["_source"]["profile_picture"]
    cur_entry["name"] = item["_source"]["name"]
    latest_array.append(cur_entry)
```

Figure 43. An example of part of the elastic_stats() function. This part creates a list of the latest Tweets.

Setup_charts(): This task is used when the '*Charts*' page is being loaded. It will also show a loading page before the data is finished being loaded. It will then display charts based on all the topics in the database. It loads data based on the past 20 days, and displays it using D3 [11].

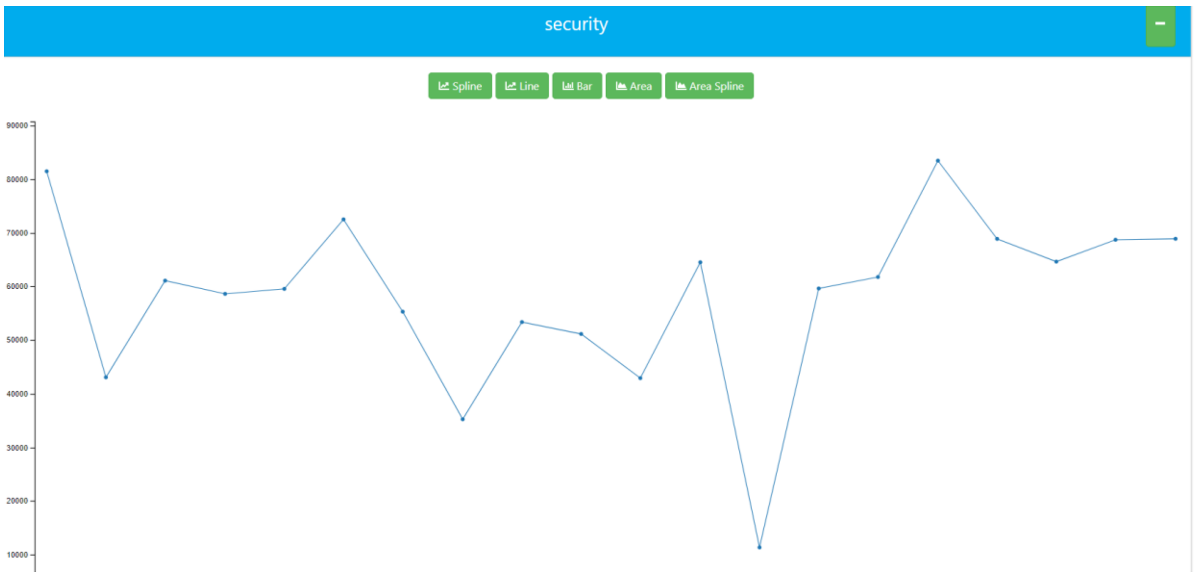
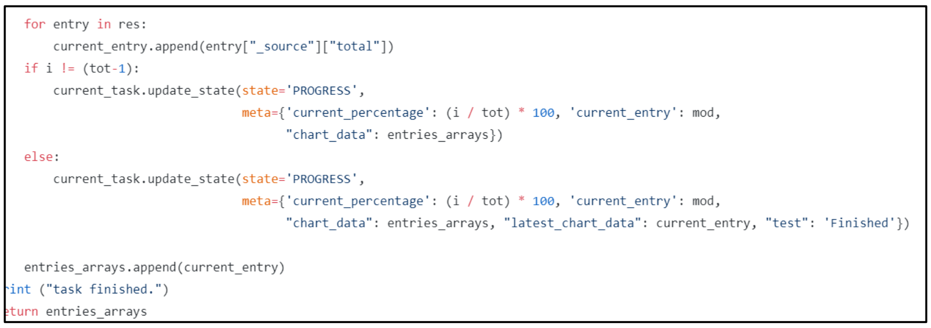


Figure 44. A sample of what the chart looks like after it has been loaded.



```
for entry in res:
    current_entry.append(entry["_source"]["total"])
if i != (tot-1):
    current_task.update_state(state='PROGRESS',
                              meta={'current_percentage': (i / tot) * 100, 'current_entry': mod,
                                    "chart_data": entries_arrays})
else:
    current_task.update_state(state='PROGRESS',
                              meta={'current_percentage': (i / tot) * 100, 'current_entry': mod,
                                    "chart_data": entries_arrays, "latest_chart_data": current_entry, "test": 'Finished'})

entries_arrays.append(current_entry)
int ("task finished.")
turn entries_arrays
```

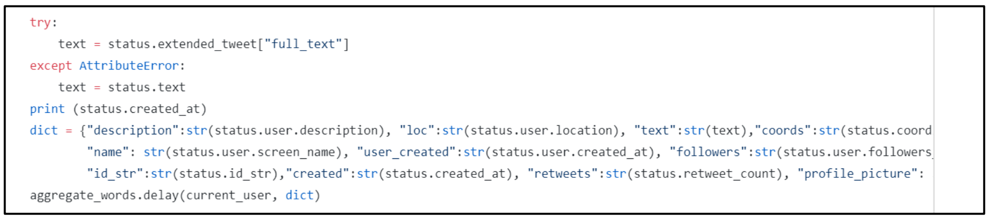
Figure 45. A sample of the code used in this setup_charts() function. It shows asynchronous requests in the update state calls and then finally returns the finished data.

6.2.4 Old_tweets queue

The last queue used by the system is the old_tweets queue which is responsible for collecting Tweets for previous days in the background. This is rather straightforward and is simply just a Celery Task that runs and aggregates to Elasticsearch with the metadata for each day just like how the live API works. Since this is the case, the code will not be discussed but can be found in the tasks.py file of the project.

7. Collecting Tweets

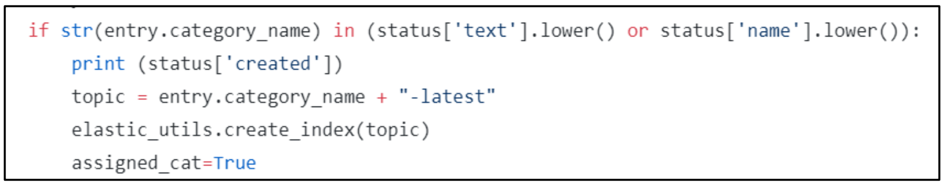
Collecting tweets is done by using the Tweepy API to set up a Stream Listener. The stream listener is used to constantly search for the latest tweets as they are posted. On startup, the Stream listener is created and is constantly running in the background via a Celery Task. The Tweets are collected, and immediately passed to the Celery queue where the Tweets are aggregated to the relevant Elasticsearch index. This is done in a queue because populating Elasticsearch involves some logic and running the logic every time, it may not be able to keep up with the stream of Tweets. By doing it this way, the stream listener only has one job and that's to pass Tweets to the queue. Another reason that passing the Tweets to a queue is helpful is because the Celery queues allow for simple multi-threading and thus will be able to process the Tweets much more efficiently.



```
try:
    text = status.extended_tweet["full_text"]
except AttributeError:
    text = status.text
print (status.created_at)
dict = {"description":str(status.user.description), "loc":str(status.user.location), "text":str(text),"coords":str(status.coord
        "name": str(status.user.screen_name), "user_created":str(status.user.created_at), "followers":str(status.user.followers_
        "id_str":str(status.id_str),"created":str(status.created_at), "retweets":str(status.retweet_count), "profile_picture":
aggregate_words.delay(current_user, dict)
```

Figure 46. This is a sample of what the stream listener is doing. It creates a dictionary of the tweet and passes it to the aggregate_words task.

After this, the task simply categorizes which topic the Tweet belongs in. It does this by searching the username and the text for the topic (All tweets collected will have one topic. If it belongs in more than one topic, the tweet will be collected twice).



```
if str(entry.category_name) in (status['text'].lower() or status['name'].lower()):
    print (status['created'])
    topic = entry.category_name + "-latest"
    elastic_utils.create_index(topic)
    assigned_cat=True
```

Figure 47. This is a section from the aggregate_words task that is responsible for assigning a specific tweet to a topic. In this case, 'entry.category_name' is the topic.

8. Reflections

On reflecting on this project, I have learned a lot about developing a project from scratch and improved my programming skills in doing so. This was the first '*proper*' project that I have done in Python and I was more used to the Groovy programming language. However, over the course of doing this project, I have developed my Python programming languages competency drastically. This isn't just about improving in Python however as I think it shows that I'd be able to do it in any programming language given enough time which is a vital skill to have in the programming industry.

For this project, I stuck to a time-management schedule to get things done and followed it well. This shows that my time-management skills have improved from doing this project. The project ended up with over 150 commits on Github with over 20,000 lines of code currently which is after removing 15,000 for better performance. I think that the numbers prove that my time-management skills have improved with how consistent the project was updated.

I learned a lot about using Git on a proper project. The approach I took to this was to do branching of features that I felt like were big changes to the project. This came to save me on some occasions where I could revert to the master branch if something went wrong with the project or if the technology didn't work (This happened with some Machine Learning approaches).

I learned about working on a big project and about how important it is to get a file structure to follow. If this isn't done, the project can start to get out of hand, and hard to find certain files. The Django approach to structuring files helped with this a lot [12]. It is also easy to see from this how important writing tests are because one small change in code can have drastic effects.

One of the main things that I learned about was Natural Language Processing(NLP) which is needed to analyse sentences, and words. One of the main things that I used for this was the natural language tool kit and the accompanying book for it [13]. The techniques that I learned about that were useful for this project were: tokenizing, stop word filtering, regular expressions, and stemming.

I learned about using Django which was quite like a technology that I already knew beforehand (Grails). From doing a project in Django it becomes clear why certain design patterns are in place and why things like a MVC (Model, View, Controller) approach can be very useful. I found some of the functionality of Django extremely beneficial in the form of Forms, Models, and in-built databases.

One of the most time-consuming and difficult things that I done in this project was using Django Asynchronously because it wasn't really built that way. Although this doesn't look that impressive in the grand scheme of the project, it is probably the thing that I am most proud of in the project because it was very difficult to implement with very limited support and it also took weeks to get functional. I learned a lot about how web sockets work, and how Django can interact with them as well as all about Django Channels, and Celery.

One thing that hasn't been discussed in detail in this report is the original approach to this problem which was using Machine Learning. I also learned a lot about doing Machine Learning in Python with the libraries scikit-learn [14], and Tensorflow [15]. The specific algorithms that I used to try attempt trend detection were the following: kmeans-clustering, LDA (Latent Dirichlet allocation), and NMF (Non-negative matrix factorization). The reason that these algorithms are no longer used is simply because when tested against the current algorithm for trend detection, they had much poorer results and it would have taken too long to get it working.

I improved my knowledge of Elasticsearch and learned a lot about making custom queries for it. I learned when to use Elasticsearch over a SQLite database and vice versa. I learned about the different approaches to Trend Detection that are being used and implemented my own custom version of one.

9. What I'd do differently & Future

If I was to do the project again, there is some things that I would change in the approach I took. I would have looked at other programming languages and frameworks other than Python and Django because at the time I was thinking of using a Machine Learning approach, and now that it has changed, it would be interesting to see the results of using other technologies for this project such as a MEAN stack approach since the asynchronous stuff is much easy to handle in something like Angular or React.

I would have paid more attention to Test Driven Development which would have saved a lot of headaches further down the line in the project.

The last thing that I would have done differently would have been not to focus on some of the smaller features first but get a functional working version that could have been improved over time.

I have a lot planned of this project that I have thought about and would like to implement. One of the main things that I would like to focus on is improving the algorithm. I would like to improve it over time and add in some Machine Learning techniques to it to see if I can make it even better. I want to do a lot more with the user interface to make the users experience better when using the application. This includes better table support, more charts, and maybe even things like phone/email notifications about the most popular trends.

I want to try and make the app not just detect keywords but be able to produce tangible sentences that would give an idea about what the topic is rather than just the keywords for the trend.

The spam detection done on this project is extremely routine and not very effective. I want to develop a better system to detect spam for the future of this project. This may also involves training a Machine Learning algorithm with what is spam and what isn't over time or maybe just a smarter algorithmic approach.

Lastly, since this project was time sensitive I only chose to do this for Twitter data. However, I would like to add more social media sites into it including Facebook, and Reddit. I think this would be cool to see how they differ in times, and results.

10. References

[1] Twitter.com. (2018). *The Guardian (@guardian) | Twitter*. [online] Available at: https://twitter.com/guardian [Accessed 17 Apr. 2018].

[2] Omnicoreagency.com. (2018). *• Twitter by the Numbers (2018): Stats, Demographics & Fun Facts*. [online] Available at: https://www.omnicoreagency.com/twitter-statistics/ [Accessed 17 Apr. 2018].

[3] Ritholtz, B. (2018). *Twitter is becoming the first and quickest source of investment news*. [online] the Guardian. Available at: https://www.theguardian.com/technology/2013/apr/23/twitter-first-source-investment-news [Accessed 18 Apr. 2018].

[4] Twitter. (2018). *Boston Police Dept. on Twitter*. [online] Available at: https://twitter.com/bostonpolice/status/323888963712606208?lang=en [Accessed 18 Apr. 2018].

[5] Blog.twitter.com. (2018). *Trend detection in social data*. [online] Available at: https://blog.twitter.com/official/en_us/a/2015/trend-detection-social-data.html [Accessed 18 Apr. 2018].

[6] Elasticsearch-py.readthedocs.io. (2018). *Python Elasticsearch Client — Elasticsearch 6.2.0 documentation*. [online] Available at: https://elasticsearch-py.readthedocs.io/en/master/ [Accessed 19 Apr. 2018].

[7] Google Cloud. (2018). *Compute Engine - IaaS | Google Cloud*. [online] Available at: https://cloud.google.com/compute/ [Accessed 19 Apr. 2018].

[8] Nltk.org. (2018). *Natural Language Toolkit — NLTK 3.2.5 documentation*. [online] Available at: https://www.nltk.org/ [Accessed 20 Apr. 2018].

[9] Channels.readthedocs.io. (2018). *Django Channels — Channels 2.1.1 documentation*. [online] Available at: https://channels.readthedocs.io/en/latest/ [Accessed 21 Apr. 2018].

[10] Celeryproject.org. (2018). *Homepage | Celery: Distributed Task Queue*. [online] Available at: http://www.celeryproject.org/ [Accessed 21 Apr. 2018].

[11] Bostock, M. (2018). *D3.js - Data-Driven Documents*. [online] D3js.org. Available at: https://d3js.org/ [Accessed 21 Apr. 2018].

[12] Django-project-skeleton.readthedocs.io. (2018). *Project Structure — django-project-skeleton 1.4 documentation*. [online] Available at: http://django-project-skeleton.readthedocs.io/en/latest/structure.html [Accessed 21 Apr. 2018].

[13] Nltk.org. (2018). *NLTK Book*. [online] Available at: http://www.nltk.org/book/ [Accessed 21 Apr. 2018].

[14] Scikit-learn.org. (2018). *scikit-learn: machine learning in Python — scikit-learn 0.19.1 documentation*. [online] Available at: http://scikit-learn.org/stable/ [Accessed 21 Apr. 2018].

[15] TensorFlow. (2018). *TensorFlow*. [online] Available at: https://www.tensorflow.org/ [Accessed 21 Apr. 2018].

References used outside report:

TrendMicro, n.d. *Data Breach.* [Online]
Available at: https://www.trendmicro.com/vinfo/us/security/definition/data-breach
[Accessed 3 December 2017].

tweepy, n.d. *Tweepy - An Easy to use Python library for accessing the Twitter API.* [Online]
Available at: http://www.tweepy.org/
[Accessed 13 November 2017].

NLTK, 2001. *NLTK home page.* [Online]
Available at: http://www.nltk.org/
[Accessed 28 November 2017].

Blog.heroku.com. (2018). *Finally, Real-Time Django Is Here: Get Started with Django Channels*. [online] Available at:
https://blog.heroku.com/in_deep_with_django_channels_the_future_of_real_time_apps_in_django
[Accessed 22 Apr. 2018].

Bl.ocks.org. (2018). *D3 Wordcloud*. [online] Available at:
https://bl.ocks.org/blockspring/847a40e23f68d6d7e8b5 [Accessed 22 Apr. 2018].

Kdnuggets.com. (2018). *Mining Twitter Data with Python Part 3: Term Frequencies*. [online] Available at: https://www.kdnuggets.com/2016/06/mining-twitter-data-python-part-3.html [Accessed 22 Apr. 2018].