Universität Innsbruck

Department of Computer Science
Research Group Quality Engineering

MASTER THESIS

The Title

Martin urek

Supervisor: Ass.-Prof. Dr. Michael Felderer

Innsbruck, June 22, 2018

Contents

Introduction

Proposed framework

proposedFramework

Social Media

Social media is a term referring to online communication channels meant for social interaction, content-sharing and collaboration. Over time, this term became widely used, its exact definition somewhat blurred and if really wanted, most of the today's websites could be labeled as social media. In context of this thesis, I refer to social media in its original meaning. To be considered a social media platform, most of following features usually need to be fulfilled:

- **User accounts** - platform allows users to create and run their own accounts that they can log into. These are online representations of their owners and serve as a too to reach and interact with other users.
- **Profile pages** - pages which represent an individual, might it be a real person, group of people or company. It should contain several personal information about the user like bio, profile picture or other personal data.
- **Friends, followers, groups** - list of accounts whose owners have some form of a relationship or common interest with the user.
- **News feeds** - Area where all new content from other connected entities appears.

Even if a platform fulfills these requirements, it doesn't necessarily have to be classified as social networking platform as pointed out in Haewoon Kwak's paper[1].

1 Potential of social media data mining

Social media are changing the way that information is passed across societies and around the world.[2] Among many other potentials of social media, there is a huge amount of data generated on daily basis. These data carry lots of real world data and if used correctly, can offer deep insight into almost any area. The process of analyzing these data and searching for repetitive reoccurring patterns with goal of predicting future trends is also called social media mining. Successful mining can not only save money and time spent on getting the data in more traditional way like surveys but can also provide crucial factor in planning or decision making of businesses. Although internet is one big hole and contains lot of false facts and desinformation, there is indication that social networks tend to favour valid information over rumours.[3]

2 Data

2.1 Getting data

As you will have a chance to see, big social networking platforms started realizing that data they own are a golden egg and getting raw full data from them got much more difficult than it was in the early years of social media age.

Twitter: When talking about sentiment analysis of social media, analysis of Twitter data has in last couple of years became almost a default choice. This change can be nicely seen in the Mantyla, Graziotin and Kuutila's [4] wordcloud of SE papers before and after 2013.0.1.

Figure 0.1: Wordcloud comparison of pre and post 2013 SE papers

To get the data from Twitter, I first tried to use the Twitter API but I very early got to know that Twitter is well aware of the worth of their data. They don't provide tweets older than 2 weeks what basically made their API inapplicable for my purposes. The only way how to get historical Twitter data is actually:

- collect them over time
- buy them from Twitter
- buy them from other companies who collect Twitter dumps over time

Therefore to obtain my data, I had to use the Twitter Search Api. To do this I used the project (https://github.com/Jefferson-Henrique/GetOldTweets-python) which provides an extra layer on top of Search Api and simplifies working with it. Using this technique, I managed to get the significant amount of data needed for all my tasks in this thesis.

This repository offers following collection of search parameters which can be used to filter specific tweets according:

- setUsername - twitter account without "@"
- setSince - lower date bound with format "yyyy-mm-dd"
- setUntil - upper date bound with format "yyyy-mm-dd"
- setQuerySearch - search text to be matched
- setTopTweets - boolean flag whether to return only top tweets
- setNear - location are reference
- setWithin - radius from "near" location
- setMaxTweets - max amount of tweets to be retrieved

I got historical twitter data by looping all projects and their release dates and requested data:

- on the release dates (Listing 1)
- in the interval of 2 days before and after release date (Listing 2)
- weekly

Listing 1: Creating command to get Tweets about a project version on release dates

```
miningConsoleCommand = "python␣Exporter.py␣--querysearch␣'" +
    frameworkName + "␣AND␣" + version + "'␣--since␣" + str(releaseDate) +
    "␣--until␣" + str(afterRelease) + "␣--output='" + frameworkName + "_"
    + str(releaseDate) + ".csv" + "'"
```

Listing 2: Creating command to get Tweets about a project version in particular time interval around release date

```
miningConsoleCommand = "python␣Exporter.py␣--querysearch␣'" +
    frameworkName + "'␣--since␣" + str(fromDate) + "␣--until␣" + str(
    toDate)  + "␣--lang␣" + "en" + "␣--maxtweets␣" + str(
    TWEETS_PER_RELEASE) + "␣--output='" + frameworkName + ".csv" + "'"
```

Facebook: I originally planned to use Facebook statuses as a big part of analyzed data. Sadly, this was not possible since Facebook Graph API doesn't allow post searching feature. There was this option till early 2014 with Facebook API 1.x versions but since Graph API has been introduced, there's no way how to make Facebook application send requests to 1.x versions of API. At first, application created before 2014 were still working on top of the early Api versions and it has been maintained but over time, all the applications were migrated to 2.x Api versions. There's currently no public way how to freely get the Facebook posts data. The other types provided by the Api are for example user, page, event, group or place.

Reddit: Despite that reddit doesn't offer big amount of user data in OSS projects subreddits, I thought getting and working with Reddit could increase the variety of users and the data which I will be working with. To get the data I used Python Reddit API Wrapper (PRAW) used to directly work with Reddit Api via HTTP requests.

Class Reddit provides a convenient way how to access Reddit API. Instance of this class can be seen as a gateway to interact with API through PRAW. To instantiate this class, user first has to register his application. This gives user unique *useragent* key which identifies the application. This is so that if your program misbehaves for some reason, it can be more easily identified, rather than look like a browser. All mandatory arguments are shown in Listing 3.

Listing 3: Instantiating Reddit class object

```
reddit = praw.Reddit(clientid='CLIENTID',clientsecret="CLIENTSECRET",
    password='PASSWORD',useragent='USERAGENT',username='USERNAME')
```

After connection to Api is successful, senting requests with PRAW is straightforward. To get the submissions from a particular subreddit in a specific time interval, just a basic loop shown in Listing 4 is enough.

Listing 4: Getting posts from subreddit

```
for submission in reddit.subreddit(redditName).submissions(FROM, TO)):
```

Each post (submission) contains among other information also array-like member variable of all comments. Simply concatenating all those gives the whole textual representation of the discussion.

Stack overflow: To extract the questions about the OSS projects of my interest I once again used provided Api. Python module called StackApi offers a way how to communicate with various Stack Exchange Api endpoints - answers, badges, comments, posts, questions, tags and users. To initiate communication with StackAPi, one needs only to specify Stack webpage and choose an endpoint, tag, time interval and if needed also some other non-mandatory parameters. Afterwards, calling *fetch* method starts returning questions. Snippet of how I got the data can be seen in Listing 5

Listing 5: Getting Stackoverflow questions with StackApi

```
SITE = StackAPI('stackoverflow')
SITE.max_pages = 1;
while True:
        questions = SITE.fetch(
                'questions',
                fromdate=date(2012, 5, 8),  # year,month,day
                todate=date(2016, 4, 15),
                tagged=project,
                filter='withbody',
                sort='creation',
                page=page
        )
```

Stack Exchange is limited on 30 requests per second what caused the process of getting the data to take much more time since program execution needed to be regularly stopped to avoid the SO throttle violation and from it resulting penalization. At first I intended to use the type posts which returns both, questions and answers, but later I've realized how huge amount of data SO contains. The average count of questions using one of the examined projects names as a tag was around 150 000. Because of this, I had to find out how to filter just the questions with higher probability of talking about bugs. Since the questions on SO do not have any labels which I could use to my advantage like I did with Git issues, I decided to keep just the questions which mention a word bug. This still left me with considerable big dataset of X questions to work with. Out of all properties of questions retrieved from questions endpoint, I decided to store and further work just with several of them mainly its title and body since these two provide most of the semantic meaning.

Open-source projects

In recent years, there has been a substantial increase in interest of open source projects. Open source software is typically developed by community of people interested in the particular area who don't necessarily know each other. These communities are usually web-based.The open-source phenomenon raises many interesting questions. Its proponents regard it as a paradigmatic change whereby the economics of private goods, built on the scarcity of resources, is replaced by the economics of public goods, where scarcity is not an issue. [5] There are many other projects which are on top of their game and yet still being open source. For example, Apache, a free server program is often a go-to decision when for a web server implementation and in 2002 it was used on 56% of web servers worldwide [6].

3 History of Open-Source Software

When talking about open source, most people immediately think of Linux. But there's so much more than than. The origin of open-source software can be traced back to the 1950s and 1960s, when software was sold together with hardware, and macros and utilities were freely exchanged in user forums. [5] In late 1983, GNU project has been announced and the next year a Free Software Foundation has been founded - both by an MIT employee Richard M. Stallman. All software written and released under Free Software Foundation had zero licences. The Linux kernel was released as freely modifiable source code in 1991 and like Unix, it attracted attention from many volunteer programmers. Another big milestone was a year 1998 when Netscape released a source code for Mozilla. In 1999 it was obvious that more and more corporate money will be invested into open source space as IBM announced their support for Linux by investing $1 billion in its development.

4 Choosing projects of interest

There are loads of OSS projects nowadays and it turned out to be pretty interesting process to choose the correct ones for my project. To make sure chosen projects fit into my work and fit all my needs I defined several requirements which needed to be fulfilled at least to some extent:

- Project needs to be widely used and well-known. This ensures there will be enough data on social media about it what will result in the less biased final results.

- Project has to have accessible bug tracking system or Git repository with list of known issues. This will provide the data for pairing the social media data with their corresponding bug items.

- Ideally, projects could be from the same area to avoid coincidentally choosing an outlier project from some either popular or unpopular field.

After considering these three points, I ended up choosing several open source web development frameworks as my projects of interest.

Project of my choice are NodeJS, AngularJS, EmberJS, VueJS for frontend technologies and Laravel, Symfony and CakePhp for backend PHP technologies. Some frameworks like Django, Meteor, React have been left out because of their misleading names would require lot of additional work to filter out data unrelated to the actual frameworks. For example, when tested Django framework, most of the twitter data referred to movie "Django Unchained".

Other very interesting group of OSS project to examine are cryptocurrencies. Being a very hot topic these days, I've decided to work with some of the most popular cryptocurrency repositories as well. These were Bitcoin, Ethereum, Litecoin, Dash and Ripple.

5 Github mining

Github as a most-popular version control system is often a go-to choice for many OSS projects. That happened to be the case also with projects I've chosen for my testing and case study in the thesis.

Thanks to Github Rest API v3[1] is data mining with Github very easy and straightforward. To send request to this API, OAuth2 token needs to be present in the request header. There are several ways how to acquire this token. I've decided to register an OAuth2 App under my GitHub account and that way I got non-changing token. Another way is to request a token programmatically, but I though it would be an unnecessary overhead. OAuth2 apps can be registered under *Account settings >Developer settings >Personal access tokens >Generate new token.*

5.1 Release dates

To get the project release dates, I've sent one request to the endpoint *git/refs/tags* (Listing 6)

[1] https://developer.github.com/v3/?

Listing 6: Requesting all project tags git api tags endpoint

```
request = Request(projectUri + "/git/refs/tags")
request.add_header('Authorization', 'token␣\%s' \% token)
project = urlopen(request).read()
tags = json.loads(project)
```

After obtaining all tags, one more request for every one of those tags was needed to get the details (Listing 7). Path to release date field withing response body as json is tag >object >url >Send new request >author >date

Listing 7: Requesting tag details and accessing release date

```
for tag in tags:
        version = tag['ref']
        got_object = tag['object']
        detailedUrl = got_object['url']

        #request details of particular release
        request = Request(detailedUrl)
        request.add_header('Authorization', 'token␣%s' % token)

        #get the person responsible for the release
        repoReleaseDetails = json.loads(urlopen(request).read())
        tagger = repoReleaseDetails['author']

        #get the date of the particular release
        releaseDate = tagger['date']
```

I saved the release dates in simple text file, one per line. This code was later on changed (in subsection 11.3) to include also number of commits per release.

5.2 Issues

To get the project issues, I've used the endpoint *issues*. It offers various parameters like state, labels, sort, direction or since date. I've decided to work just with closed issues and snippet where I'm sending the request can be seen in listing 8).

Listing 8: Requesting 100 closed issues

```
request = Request(projectUri + '/issues?state=closed&perpage=100&page=' +
    str(pageNum))
```

It is also worth noting here that GitHub's REST API v3 considers every pull request an issue so I had to identify and filter them out using their *pullRequest* key.

During my later work, I've realized the ammount of issues is just too big and broad and not every issue is a bug or even remotely similar to bug. That was when I have decided to filter the issues and keep only real bugs. For this I've used Git labels. Labels on GitHub help organize and prioritize work. They can be applied to issues and pull

requests to signify priority, category, or any other information you find useful. There are two types of labels - default and custom. GitHub provides default ones in every new repository. All default labels can be seen in table 0.2 and can be used to create a standard workflow in a repository:

Label	Description
bug	Indicates an unexpected problem or unintended behavior
duplicate	Indicates similar issues or pull requests
enhancement	Indicates new feature requests
good first issue	Indicates a good issue for first-time contributors
help wanted	Indicates that a maintainer wants help on an issue or pull request
invalid	Indicates that an issue or pull request is no longer relevant
question	Indicates that an issue or pull request needs more information
wontfix	Indicates that work won't continue on an issue or pull request

Figure 0.2: Default Git labels provided for every repository

From there I have chosen the label *bug*. Then I have checked all custom tags of all repositories and chosen just those which were semantically similar to bug. All chosen labels can be seen in the table 0.1

Repository	**Chosen custom labels**
NodeJS	confirmed-bug, errors
AngularJS	type: bug
VueJS	browser-quirks, 1.x, 2.x
Aurelia	enhancement
EmberJS	Bug

Table 0.1: Reddit submissions counts

Sentiment analysis

SE and opinion mining is the field of study that analyzes people's opinions, sentiments, evaluations, attitudes, and emotions from written language.[7] It's also known as emotion AI or opinion mining. The main and basic task of this field is to correctly classify the polarity of a particular text and evaluate whether is it positive, negative or in some cases neutral. It can be used in almost any situation where data need to be analysed for its sentiment aspect, what means the application options are almost endless. for example product reviews, online discussions or social media content.

SE is in demand mostly because of its efficiency. Tens of thousands of documents can be analysed within seconds and despite results are not always as exact as human workers would produce, the efficiency boost is often too big not to take advantage of it.

SE can be divided into several steps:

1. **Data collection** - involves all the substeps required to gather user-generated content from any source. Surveys, blogs, various forums and social media. These all contains huge amount of real people from real world with real experiences. These data are always expressed completely different - using slang, shortcuts, internet language or generally just being used in different context.

2. **Data preprocessing** - this step represents cleaning the data. Data preprocessing can often have a significant impact on generalization performance of a supervised ML algorithms [8] and if there is much irrelevant information and data are unreliable, then knowledge discovery during the training phase is more difficult. It removes irrelevant content which could potentially lead to bad and incorrect results. This is a very delicate step because it manipulates the raw data and if not done correctly, it can easily change results.

3. **Sentiment detection** - this step basically stands for training of the classifier. Subjective feelings and expressions are highlighted, emphasized and retained while objective information like facts are discarded and ignored.

4. **Sentiment classification** - subjective expressions are classified.

5. **Output interpretation** - graphical presentation of obtained results. Time and sentiment can be analyzed to construct various charts, graphs, timelines and many other metrices.

6 History

Interest in opinion of other individuals is probably as old as the communication itself. There is evidence that already in ancient Greece, generals were trying to detect dissent among their subordinates using various "primitive" approaches [9]. Another approach to measure and evaluate a public opinion coming from ancient Greece and used to these date is voting. In the first decades of twentieth century, efforts in capturing public opinion started utilize questionaires and in 1937, first scientific journal on public opinion was founded.

In the last 10 years, SE and ML in general experienced a big boom. According to data collected by Mantyla, Graziotin and Kuutila [4], nearly 7000 papers about sentiment analysis have already been written and not surprisingly, 99% of them were published after 2004 - making sentiment analysis one of the fastest growing research areas. The increasing interest about this area as published by Mantyla, Graziotin and Kuutila [4] can be seen in the graph 0.3.

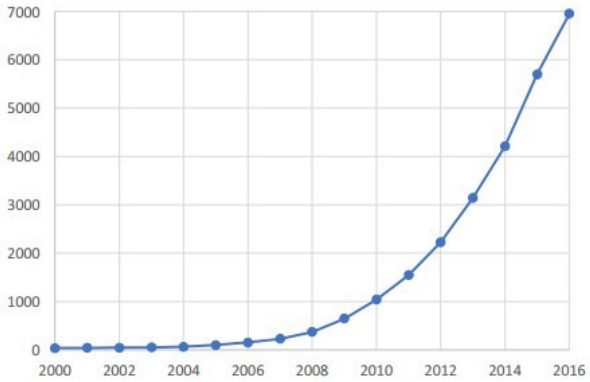


Figure 0.3: Cumulative count of papers about sentiment analysis

7 Training datasets

One of the main building blocks of any correctly and accurately functioning ML projects is a training phase. It can actually be seen as a base for the whole project. One can have the best fine-tuned optimized classifier, but if the training data he used do not fit the domain where the classifier is intended to be used, results of the classifier can be (surprisingly) bad. It's the same as house and its base. If the base is not done correctly, however cool architectural solution have other storeys used, house is still going to fall in the next big storm.

That's why choosing a sufficient and fitting dataset to train my classifiers was a very important task. The datasets I've considered were:

- **Dataset140**[2] - it is currently the biggest dataset with tweets labeled by their sentiment. What is interesting and makes the dataset special is that opposed to other datasets being manually annotated by humans, this one was created by a program. It contains 1.6 million tweets with their polarity score (0 = negative, 2 = neutral, 4 = positive), tweet id, date of tweet publication, author of the tweet and the text of the tweet. More about how this dataset was created can be found in Go et al. paper [10]
- **Movie review data**[3] - Thousand positive and thousand negative labeled movie reviews. This dataset was introduced in Pang/Lee ACL 2004 [14]
- **Hu-Liu lexicon**[4] - plain list of 6800 common English words labeled as positive and negative
- **Warriner et al lexicon**[5] - This list of words was collected with Amazon Mechanical Turk. Three components of emotions are traditionally distinguished: valence (the pleasantness of a stimulus), arousal (the intensity of emotion provoked by a stimulus), and dominance (the degree of control exerted by a stimulus) [11]. Warriner and Kuperman extended ANEW norms collected by Bradley and Lang from 1034 words to 13,915 words (lemmas).
- **Stack overflow dataset**[6] - Later into the thesis I've decided to test dataset of 1500 manually labeled Stack overflow sentences created by Bin Lin et al. in their late paper on negative results in SA called "How far can we go"
- **My own cryptocurrency tweets dataset** - As already said before, performance of the classifier depends on how close the training data are to the real use-case data. That's why I considered and even started to create my own dataset targeting specifically only cryptocurrency tweets.

8 Language processing tools

From the very beginning, I knew I wanted to use Python. I had some slight background knowledge in ML from online courses and most of them were done in Python. Therefore while searching and deciding which library should I use, I've always given a slight edge to the Python options. I've considered (and tested) these:

- **NLTK** - probably the best-known Python module for NLP. It provides easy-to-use interfaces for more than 50 corpora and lexical resources. It also offeres a rich palette of processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning.

[2] http://cs.stanford.edu/people/alecmgo/trainingandtestdata.zip
[3] http://www.cs.cornell.edu/people/pabo/movie-review-data/
[4] https://github.com/woodrad/Twitter-Sentiment-Mining/tree/master/Hu%20and%20Liu%20Sentiment%20Lexicon
[5] http://crr.ugent.be/archives/1003
[6] https://sentiment-se.github.io/replication.zip

- **Textblob: Simplified Text Processing** - as a name says, Textblob provides easy processing and is actually built on top of NLTK. It provides a simple API for diving into common natural language processing (NLP) tasks such as part-of-speech tagging, noun phrase extraction, sentiment analysis (Naive Bayes, Decision Tree), classification, translation and more.
- **Scikit-learn** - Python module for general ML, data mining and data analysis. It is built on NumPy, SciPy and matplotlib modules.

Also, sentiment analysis is just one part of the task. To evaluate the data and find pattern, basic data science algorithms will be needed. With data science, R is very often listed as a default choice. Therefore were the results of google trends query shown in Figure pretty surprising. This definitely helped my decision with sticking to Python.

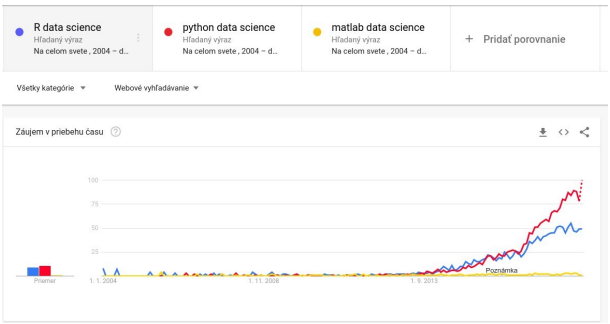


Figure 0.4: Google trend of searches regarding data science with various programming languages

9 Performance metrics

Terminology Before diving into talk about the metrics, there are 4 crucial terms which need to be explained:

- **Tue Positives (TP)** - instances correctly labeled as positive
- **True Negatives (TN)**- instances correctly labeled as negative
- **False Positives (FP)** - instances incorrectly labeled as positive
- **False Negatives (FN)**- instances incorrectly labeled as negative

The metrics that you choose to evaluate your machine learning algorithms are very important and not all are suitable for every situation. Choice of metrics influences how the performance of machine learning algorithms is measured and choosing a wrong evaluation metric for particular use could potentially lead towards eliminating the best performing algorithm in favour of the worse one. Here are some of the most often used metrics used to evaluate classification algorithms. It's also useful to choose the metric

before doing the analysis, so you won't get distracted by already having the results in case of doing the decision later.

- **Classification accuracy** - this is the most intuitive and common evaluation metric for classification problems but it is also the most misused one. It is really only suitable when there are an equal number of observations in each class (which is rarely the case) and that all predictions and prediction errors are equally important, which is often not the case. In case of imbalanced dataset with 9% of instances in one class and only 10% in the other, predicting every instance as a majority class without even considering its features would lead to high accuracy of 90%. This is called **accuracy paradox.**

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Confusion matrix** - clean and unambiguous way to present the prediction results of a classifier. If the classification is binary (there are only 2 classes), this matrix has 2 rows and 2 columns - therefore altogether 4 cells which are filled with true/false positives/negatives count. Such scenario is demonstrated in table 0.2. Although the confusion matrix shows all of the information about the classifier's performance, more meaningful measures can be extracted from it to illustrate certain performance criteria.[**?**].

Confusion matrix		
	Predicted positive	Predicted negative
Real positive	TP	FN
Real negative	FP	TN

Table 0.2: Confusion matrix

- **Precision** - Precision can be seen as a representation of a classifiers exactness. A low precision can also indicate a large number of False Positives. If the precision is high, it says that there's a high probability of positive label being True Positive. It cannot be tricked but it also hides a lot.

$$Precision = \frac{TP}{TP + FP}$$

- **Recall** - also called *sensitivity* or the *True Positive Rate*, it is a number of True Positives divided by the number of True Positives and the number of False Negatives. In other words, it is a ratio of how many of all positive instances have been identified. Recall can be tricked (labeling all as majority class) but if used next to precision, it gives extra information

$$Recall = \frac{TP}{TP + FN}$$

- **F1 score** - as already mentioned, precision hides some facts and recall can be tricked. To give the full story, they need to be used together. That's what F1 measure (F measure) is for. It is the harmonic average of the precision and recall, where its best value is at 1.

$$F = 2 * \frac{precision * recall}{precision + recall}$$

Nice example to demonstrate difference between precision and recall is the concept of Indian Jurisprudence, where "100 culprits may let go free but no innocent should be punished". If we let go so many culprits in order to ensure no innocent is punished, recall will be pretty low, but precision very high.

10 Data to analyze

11 Results

11.1 Performance

11.2 Cross-correlation between releases and sentiment change

When working with time series, we often want to determine whether one series causes changes in another. To find this relationship, measuring a cross-correlation and finding a lag is one way how to do it. Lag represents when change in one data series transfers to the other several periods later.

To ensure a cross-correlation calculation makes sense, first I have to determine, whether are the data stationary. A stationary time series is one whose properties do not depend on the time at which the series is observed[12]. More precisely, if y_t is a stationary time series, then for all s, the distribution of $(y_t,,y_{t+s})$ does not depend on t.

To determine whether my data are stationary, I've used the Dickey-Fuller test method of tseries package in R. Results can be seen in the table 0.3 and 0.4

Stationarity test of web frameworks sentiment data		
Framework	Dickey-Fuller	p-value
NodeJS	-2.6775	0.2964
AngularJS	-3.883	0.0199
EmberJS	-4.0783	0.0199
VueJS	-3.438	0.0646
CakePHP	-3.480	0.04847
Laravel	-2.57	0.3431
Symfony	-4.3979	0.01

Table 0.3: Stationarity test of sentiment

Stationarity test of web frameworks release count		
Framework	Dickey-Fuller	p-value
NodeJS	-2.896	0.205
AngularJS	-2.547	0.353
EmberJS	-3.297	0.0802
VueJS	-2.158	0.511
CakePHP	-3.224	0.08915
Laravel	-2.368	0.425
Symfony	-2.218	0.488

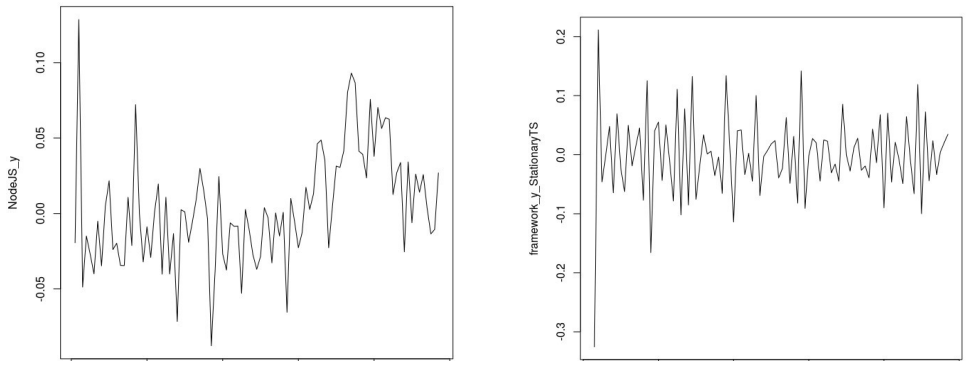
Table 0.4: Stationarity test of release counts

As we can see, p-values are always higher than 0.05 what indicates non-stationarity of the data, therefore I can't calculate the cross-correlation on them in this state. To transform non-stationary data into stationary, 2 approaches can be used. These are differencing and transforming. I've taken data series and differenced the values in listing 9. I've executed both, seasonal differencing and stationary differencing although seasonal probably was not needed because the data should not be dependant on the season.

Listing 9: Used differencing method in R

```
Differencing <- function(x,y)
{
 framework_x_seasdiff <- diff(x,differences=1)  # seasonal differencing
 framework_x_Stationary <- diff(framework_x_seasdiff, differences= 1)
 framework_y_seasdiff <- diff(y, differences=1)
 framework_y_Stationary <- diff(framework_y_seasdiff, differences= 1)
 return(list(framework_x_Stationary,framework_y_Stationary))
}
```

New differenced values do appear to be stationary in mean and variance, as the level and the variance of the series stays roughly constant over time. Sentiment for NodeJS before and after differencing can be seen in Figure 0.5



(a) Before differencing (b) After differencing

Figure 0.5: NodeJS monthly sentiment values

Same procedure needed to be done with the "number of releases per month" data and afterwards. Then, cross-correlation could be executed. For this task I've used ccf method in R which implements Pearson's correlation calculation method. Results for all 7 OSS projects can be seen in Figure 0.6

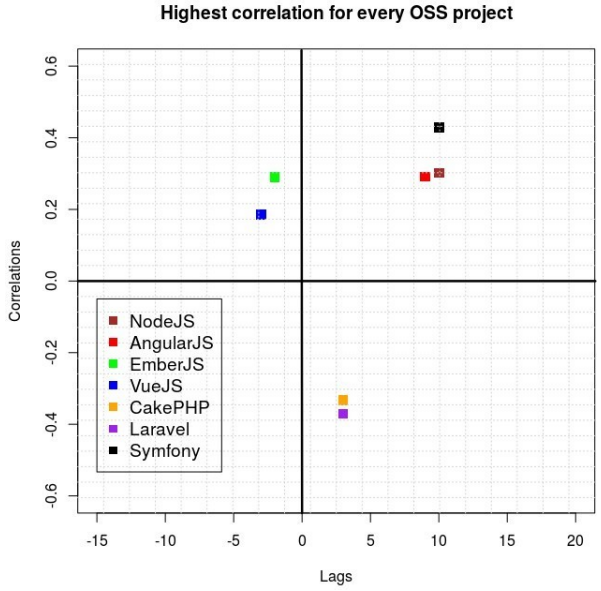


Figure 0.6: Highest correlations for every OSS project

Results interpretation: As we can there is no general pattern. Maximal project correlations happen to occupy 3 of 4 possible quadrants. Each quadrant represents a different relationship between number of releases and sentiment change.

- **I. Quadrant**(Positive correlation + positive lag) - Increase of release count increases a sentiment
- **II. Quadrant**(Positive correlation + negative lag) - Increase of sentiment increases a release count
- **III. Quadrant**(Negative correlation + positive lag) - Increase of release count decreases a sentiment
- **IV. Quadrant**(Negative correlation + Negative lag) - Increase of sentiment decreases a release count

Also, in Figure 0.7 are the results without making the data series stationary. It's obvious that making the data stationary has a big impact on the results.

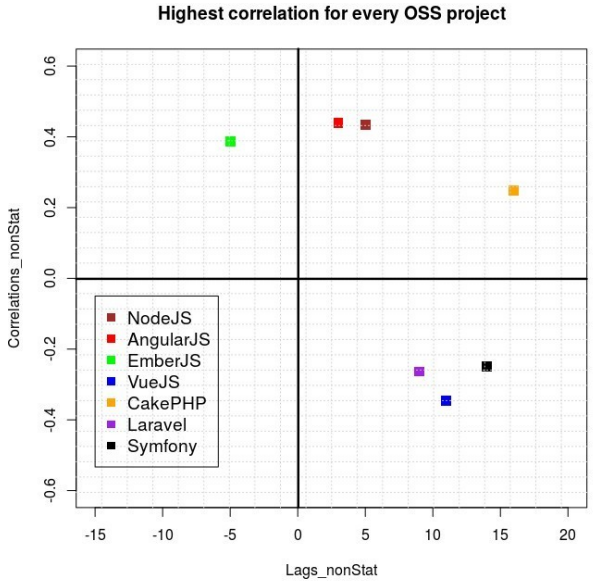


Figure 0.7: Highest correlations for every OSS project (non-stationary)

11.3 Commits count within releases

Initially, I thought that to modify project to take into account a size of the release (amount of commits) will be pretty straightforward task. It actually was straightforward, but as always I've encountered several unexpected problems on the way.

I intended to extend my previously used method from section 5.1 which uses Git Api tags endpoint to get the release dates. Unfortunately I wasn't able to find number of commits in the returned objects. JSON object returned from API has following structure:

```
{
  "url": X,
  "assets_url": X,
  "upload_url": X,
  "html_url": X,
  "id": X,
  "tag_name": X,
  "target_commitish":X,
  "name": X,
  "draft": X,
  "author":{},
  "prerelease": X,
  "created_at": X,
  "published_at": X,
  "assets":[],
  "tarball_url": X,
  "zipball_url":X,
  }
```


I've done some extra searching but didn't want to spend extra time so I've decided to go the way I knew will work. Instead of using Api to get the commit counts, I've crawled Github UI page of each release and extracted information directly from page source code. Each release details page provides information how many commits behind the current HEAD the commit is. The difference in this number between two following releases represents count of new commits for a release. Results of simple tabular substraction with spreadsheet formula needed to be manually corrected because projects often release several branches parallel and therefore substraction from the previous release was not always the correct one.

Eventually, I got correct number of commits for every release and could execute the same cross-correlation analysis described in the previous chapter, but this time instead of releases count, I've explored relationship between sentiment and commits count. One possible flaw in the commit count data are the pre-releases. I treated them as normal releases because they do offer new features but those very same commits are then counted in the official relaeses later on.

After getting the data ready I performed a stationarity test for commit counts. Sentiment values ar the same as before with count of releases. Results can be seen in table 0.5

Stationarity test of web frameworks commit counts		
Framework	Dickey-Fuller	p-value
NodeJS	-7.0239	0.01
AngularJS	-2.547	0.3531
EmberJS	-3.2764	0.0831
VueJS	-2.9748	0.1886
CakePHP	-3.655	0.03283
Laravel	-2.919	0.2084
Symfony	-4.8461	0.01

Table 0.5: Stationarity test of commit counts

I see that there are again several data series (AngularJS, EmberJS, VueJS, Laravel + NodeJS because of unstationarity of sentiment data) which are not stationary so exactly as before with release counts, I had to transform the data. After that, Pearson's cross correlation was calculated. Results for all 7 OSS projects can be seen in Figure 0.8

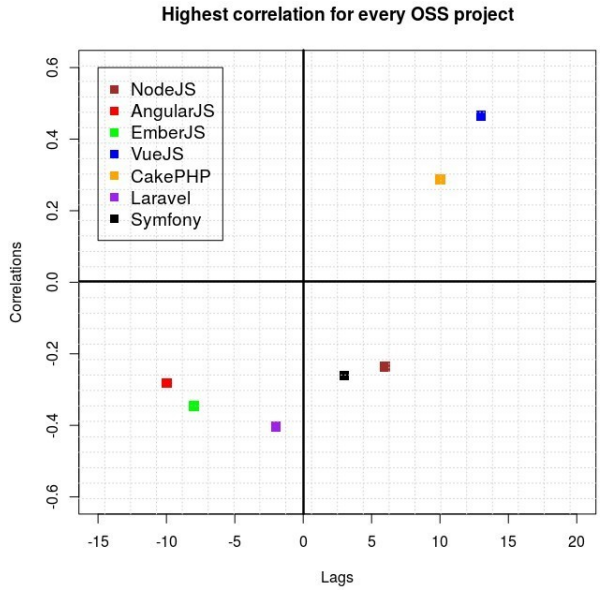


Figure 0.8: Highest correlations for every OSS project

If I would skip the step of making the data stationary, results would again look completely different 0.9.

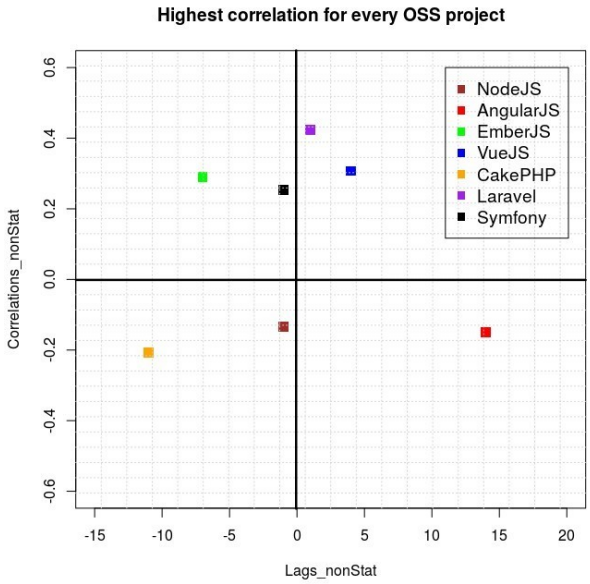


Figure 0.9: Highest correlations for every OSS project (non-stationary)

Linking bug repositories and social media

My goal of pairing bugs with social media might be somewhat similar to nowadays very active field of bug report duplicate discovery but has also a lot common with topic modeling and general text similarity algorithms.

To link any two texts based on their content, there's an obvious need for understanding what the text features are. To do this, there are several ways how to calculate text (string) similarity value or one can even execute so called "Topic modeling" algorithm and try to connect documents based on their matching topics.

Topic modeling is a method used to organize and summarize large textual information. It's used to discover hidden topical patterns and annotate documents according to these topics. It can also be described as a method of finding group of words (i.e topic) from in a text that best represents the information in the collection.

The obstacle of using a topic modeling for my case is that neither SO nor Reddit questions are not long enough. This could potentially be avoided by concatenating the whole discussions into one long text but these are still too topic-specific to get reasonable output. In this case, topic modeling output is not granular enough to differentiate among similar texts which are all from the very same domain.

Because the output provided by topic modeling wasn't enough to pair particular items, I searched for, found and considered several alternative approaches.

12 Aproaches

There are many ways and approaches how to find out whether 2 texts are similar or not. Most of them are to some extent very similar as the general need says some textual features of need to be extracted and compared.

I've tried following algorithms:

1. Nouns extraction
2. String similarity using NLTK
3. String similarity using Gensim

4. The textual properties, title and description are concatenated together and then used to numeric features using the simple TakeLab system. This approach was shown by Lazar, Ritchey and Sharif. [**?**]

13 Available data

Git issue reports: Using the approach described in the section 5.2, I've downloaded 96,651 issue reports from Git while 25,978 of those labeled as bug or similar. Because this part of the thesis was more just a PoC than the part with sentiment analysis, I've decided not to work with all the data and rather just picked several projects of interest. The bug counts among this projects is displayed in the table 0.6.

Framework	**Bug count**
NodeJS	1615
AngularJS	2225
EmberJS	1284
VueJS	353
Aurelia	73
Bower	155

Table 0.6: Bug count per project

Stack overflow questions: SO mining has been described in section 2.1 and I've downloaded 5,847 questions. There are thousand questions for AngularJS, NodeJS, Bower, Ruby on Rails and VueJS each and EmberJS has only 847 questions. Downside is, that despite having a lot of questions, it doesn't necessarily mean that each and every one of them talks about some known bug. Actually, opposite is true as out of all those questions only very tiny percentage does (can be seen in table 0.7). Because the projects I worked with till now had just so few questions linked to its own issues, I had to increase the number of bugs in the dataset. Since StackApi is limited to 10 thousand requests per day, I've decided to from opposite end and crawled through all closed git issues of projects and checked if there's a SO question linked to them. If that was the case I've saved both, git issue and SO question and that way increased the size of my dataset (can be seen in table 0.8)

Framework	Question count	Bug count
NodeJS	1000	2
AngularJS	1000	0
EmberJS	847	2
VueJS	1000	0
Aurelia	646	4
Bower	1000	5

Table 0.7: Bug count per project

Framework	Bug count
NodeJS	2
AngularJS	46
EmberJS	20
VueJS	1
Aurelia	3
Bower	4
Django	13

Table 0.8: Extra bugs added to the dataset

Reddit dialogues: Reddit subreddits mining has been described in the same subsection as SO mining. After I have learnt that there is too much data online, with Reddit I downloaded only submissions with any Git issue link (not necessarily own issue). Results of this process are shown in the table 0.9.

Framework	Submissions count
NodeJS	108
AngularJS	43
VueJS	20
EmberJS	13

Table 0.9: Reddit submissions counts

14 Similarity results

Linking items and in general finding similarity among these already area-specific texts proved to be a problematic task.

14.1 NLTK

Stack Overflow: The average similarity between SO questions talking about particular issue and that particular issue body description is 0.316 without body preprocessing and 0.292 with body preprocessing.

The distribution of similarities in buckets by increased by 0.05 can be seen in histogram in Figure 0.10

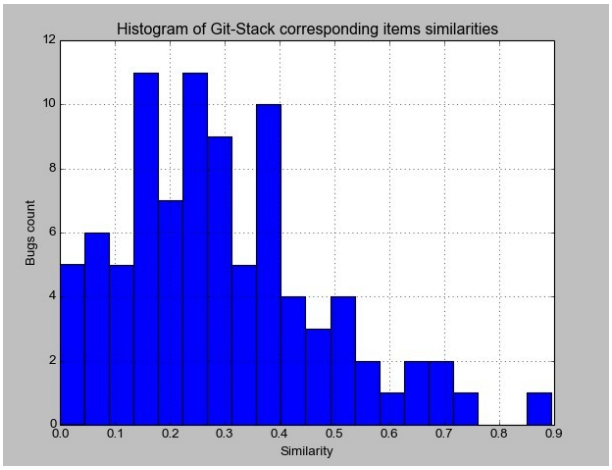


Figure 0.10: Histogram of similarities distribution among git issues and their matching SO questions

For random SO questions, amount of comparisons to Git issues needed to be limited. If every SO question would be compared to every Git issue, time of computation would exceed timeframe of this thesis. Every SO question was therefore compared to 3 random git issues and resulting average similarity scores are in following figures.

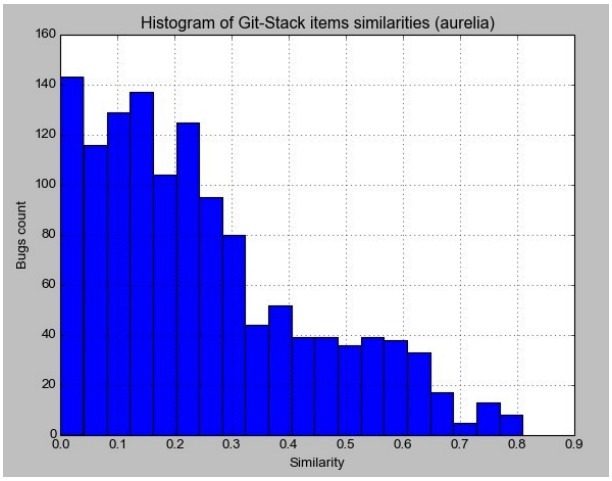
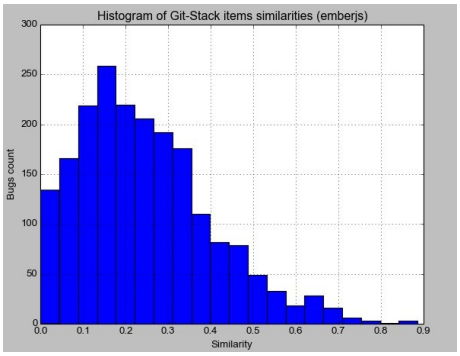
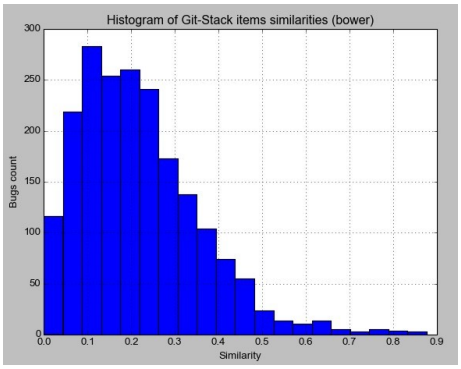


Figure 0.11: Histogram of Aurelia SO questions and random git issues. Average similarity was 0.244

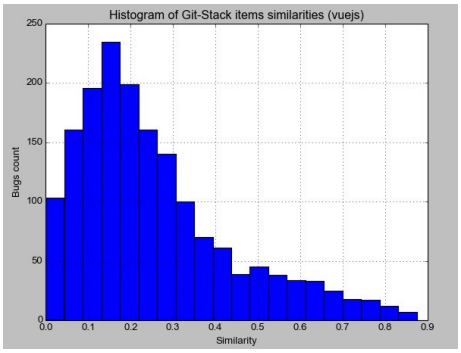


(a) EmberJS average similarity - 0.247

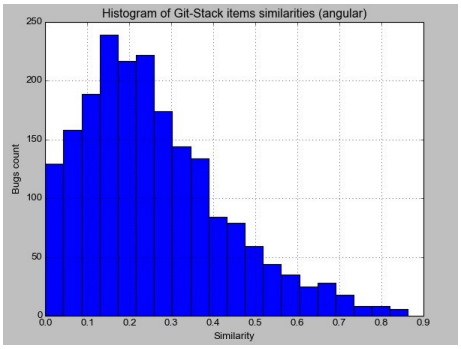


(b) Bower average similarity - 0.217

Figure 0.12: EmberJS and Bower similarity histogram



(a) VueJS average similarity - 0.255



(b) AngularJS average similarity - 0.258

Figure 0.13: VueJS and AngularJS similarity histogram

Comparing Git bugs descriptions with SO questions talking about own project issues and general issues, similarity values are following 0.10. Same table but for Reddit discussions can be seen in 0.11

Framework	**Own issues similarity**	**All issues similarity**
NodeJS	0.265	X
Angular	0.241	X
EmberJS	0.282	X
VueJS	X	X

Table 0.10: NLTK similarity values for SO questions

Reddit: Here I've calculated the similarity between the bug description and either particular comment in the reddit discussion which mentioned the bug or the whole discussion itself. Average similarity score for all considered projects (NodeJS, AngularJS, VueJS and EmberJS) was 0.481 for the whole discussion and 0.396 for the comment itself. Detailed scores for each project can be found in table 0.11. Subreddit for EmberJS didn't reference any of its own bugs.

Framework	**Bug comment**	**Whole discussion**
NodeJS	0.447	0.507
AngularJS	0.306	0.57
EmberJS	X	X
VueJS	0.359	0.380

Table 0.11: Reddit NLTK similarity values

This indicates that the semantic meaning of the bug is better expressed in the whole discussion rather than just the particular comment which referenced the bug. This

made me question if it could be generalized that longer the text is, more similar it is to actual bug description. I've plotted a relationship between similarity score and length in Figures 0.15 and 0.14.

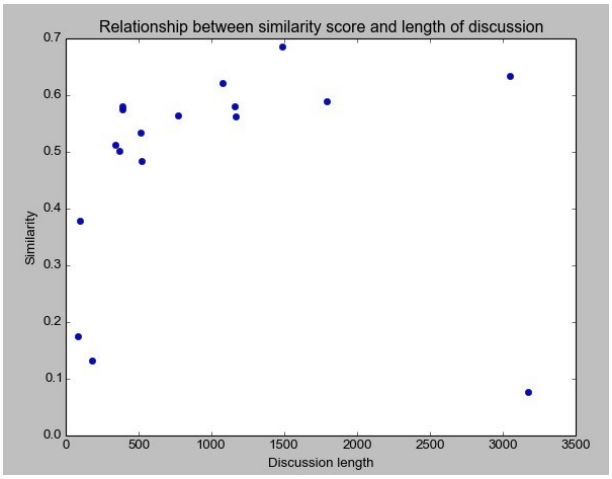


Figure 0.14: Discussion lengths and similarity scores with the issue

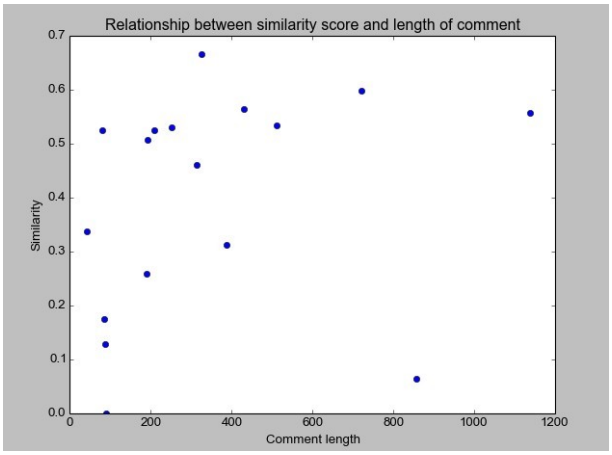


Figure 0.15: Comment lengths and similarity scores with the issue

15 Buckets

ReBucket measures the similarities of call stacks in crash reports and then assigns the reports to appropriate buckets based on the similarity values.[**?**]

16 Using GIT labels

One more considered approach how to recognize an issue's topic was using the GIT labels.

These default labels come as a big help in directing the project and targeting the most important issues, but they don't say much about the nature of the issue itself.

The custom tags tell are used to specify the part of the project, where the issue is located but they still don't give any semantic information about the issue itself. That's the reason why this approach was rejected.

Bibliography

[1] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.

[2] Antony Mayfield. What is social media. 2008.

[3] Carlos Castillo, Marcelo Mendoza, and Barbara Poblete. Information credibility on twitter. In *Proceedings of the 20th international conference on World wide web*, pages 675–684. ACM, 2011.

[4] Mika V Mäntylä, Daniel Graziotin, and Miikka Kuutila. The evolution of sentiment analysisa review of research topics, venues, and top cited papers. *Computer Science Review*, 27:16–32, 2018.

[5] Shaosong Ou Alexander Hars. Working for free? motivations for participating in open-source projects. *International Journal of Electronic Commerce*, 6(3):25–39, 2002.

[6] Josh Lerner and Jean Tirole. The open source movement: Key research questions. *European economic review*, 45(4-6):819–826, 2001.

[7] Bing Liu. Sentiment analysis and opinion mining. *Synthesis lectures on human language technologies*, 5(1):1–167, 2012.

[8] SB Kotsiantis, D Kanellopoulos, and PE Pintelas. Data preprocessing for supervised leaning. *International Journal of Computer Science*, 1(2):111–117, 2006.

[9] JA Richmond. Spies in ancient greece. *Greece & Rome*, 45(1):1–18, 1998.

[10] Alec Go, Richa Bhayani, and Lei Huang. Twitter sentiment classification using distant supervision. *CS224N Project Report, Stanford*, 1(12), 2009.

[11] Amy Beth Warriner, Victor Kuperman, and Marc Brysbaert. Norms of valence, arousal, and dominance for 13,915 english lemmas. *Behavior research methods*, 45(4):1191–1207, 2013.

[12] Rob J Hyndman, George Athanasopoulos, Slava Razbash, Drew Schmidt, Zhenyu Zhou, Yousaf Khan, Christoph Bergmeir, and Earo Wang. forecast: Forecasting functions for time series and linear models, 2013. *R package version*, 5.

[13] Joseph Feller, Brian Fitzgerald, et al. *Understanding open source software development*. Addison-Wesley London, 2002.

[14] Bo Pang and Lillian Lee. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *Proceedings of the 42nd annual meeting on Association for Computational Linguistics*, page 271. Association for Computational Linguistics, 2004.