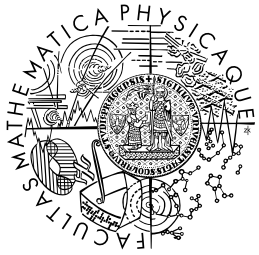


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Samuel Fanči

Social Networks: Analysis of Evolution and Sentiment

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: doc. RNDr. Iveta Mrázová, CSc.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I want to thank my supervisor doc. RNDr. Iveta Mrázová, Csc. for her endless patience and invaluable insight during the work on this thesis. I would also like to thank my family, for giving me the opportunity to study abroad and always supporting me. Finally, I want to thank my girlfriend and my friends, for being there when I needed them most.

Title: Social Networks: Analysis of Evolution and Sentiment

Author: Samuel Fanči

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: doc. RNDr. Iveta Mrázová, CSc., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Nowadays, social networks form an essential part of our lives. Their analysis helps us better understand various social phenomena, identify individuals influencing society, and model future developments of communities. Often, real-world social networks conform to power-law degree distribution. We oriented our research toward investigating communities surrounding two well-known companies: GameStop and Enron. Using the data obtained from Reddit and Twitter, we have trained machine learning models like Support vector machines and Neural networks to assess the sentiment of the GameStop community. The results confirm the expected positive sentiment following the GameStop price spike in 2021.

We constructed the respective social networks based on the available datasets and identified their vital individuals according to selected centrality measures. Publicly known figures like Ryan Cohen in the case of GameStop and Jeff Skilling in the case of Enron are ranked high according to PageRank and Authority scores. On the other hand, minor influencers from the GameStop community and the upper management of Enron were assigned top ranks of the Hub score and Betweenness centrality. A statistical analysis using the goodness-of-fit test for the power-law degree distribution was performed for both networks. Results indicate a plausible fit only for the in-degree distribution of the Twitter network and for the in- and out-degree distributions of the Enron network ($p = 0.8366$, $p = 0.496$, and $p = 0.546$, respectively).

Keywords: social networks, sentiment analysis, data mining, detection of influential individuals, machine learning

Contents

Introduction

Social networks have become an inseparable part of our daily lives and represent the focus of this thesis. By studying them, we can better understand social dynamics, explain events happening in our world, and possibly predict future societal developments. We can model the spread of infectious diseases in a population, the social impact of certain actors in the network, or estimate the results of upcoming elections. Combining the knowledge from the fields like graph theory, mathematical analysis, and artificial intelligence, we will analyze real-world social networks formed around two well-known companies: GameStop and Enron.

GameStop is an American video game retailer that made waves in the news throughout 2021. The company's stock value was slowly decreasing in the previous years since its online competition overtook the business. However, at the end of January 2021, the value of the ailing company skyrocketed, briefly reaching $483, up from around $17 a few weeks prior. This spike in its value affected a community of retail investors/gamblers on Reddit, called r/wallstreetbets, and later on Twitter. Obviously, the value of GameStop rose due to hype, regardless of its actual value.

Enron was an American energy company that diversified into many sectors, including security trading. It is infamous for being one of history's most prominent corporate fraud cases. Much of the upper management was complicit in the crime. The law enforcement agencies subpoenaed the internal company emails and released them publicly.

In the case of GameStop, it would be interesting to see the evolution of sentiment of the communities toward GameStop following the price spike. Creating a sentiment model will be our first goal. Its purpose will be to classify the provided input text into one of three categories: positive, negative, or neutral (w.r.t. GameStop). To address this issue, we will use selected machine learning models of varying complexity: Naive Bayes, Support Vector Machines, Random Forests, Recurrent, and Perceptron-like Neural Networks.

Our second goal will be identifying the key players in the investigated communities (both GameStop and Enron). We would like to see if the most talked about people regarding these companies are also the structurally most important ones. For this purpose, we will construct a social network from each dataset and, for each node, calculate multiple importance scores based on different importance measures. These measures will be PageRank - for general structural importance, HITS Authority and Hub scores - measuring how well a node functions as a source and a hub for information spread, respectively and, lastly, Betweenness centrality - measuring the criticality of a node w.r.t. the flow of information throughout the network.

Many real-world social networks follow the power-law degree distribution. There are several exciting consequences to networks with this degree distribution, for example, extreme disparities between node degrees. Our third goal will therefore be to fit a power law for both of these networks and determine the statistical significance of this fit.

The thesis consists of five chapters. The first chapter contains the used notations and definitions of mathematical notions we will use in the latter parts

of the work. In the second chapter, we will lay down the theoretical foundations for studying social networks, discuss some of their properties, and describe the previously mentioned node importance measures. We will also explain the goodness-of-fit statistical test for the power-law fit. The third chapter provides an overview of the machine learning models used for sentiment analysis and some preprocessing techniques used for their training. The fourth chapter includes a preliminary analysis of the gathered Twitter, Reddit, and Enron data, together with hypotheses for the outcomes of experiments. Finally, the fifth chapter describes the experiments we have performed and the results we have obtained.

1. Definitions and notation

In this chapter are defined some basic notions used throughout the thesis, along with definitions of basic mathematical concepts.

1.1 Notation

- c - a scalar number
- $\mathbf{v} \in \mathbb{R}^n$ - a column vector of scalars $\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_n \end{pmatrix}$
- $\mathbf{A} \in \mathbb{R}^{n \times m} = \begin{pmatrix} a_{1,1} & a_{1,2} & ... & a_{1,m} \\ . & & & . \\ . & & & . \\ . & & & . \\ a_{n,1} & a_{n,2} & ... & a_{n,m} \end{pmatrix}$ - a matrix consisting of real numbers containing n rows and m columns
- $\mathbf{A}_{i,j}$ - the value at the i-th row and j-th column in matrix $\mathbf{A}$

1.2 Definitions

These definitions come from Chapter 1 and 2 of [1].

Definition 1 (Probability space)**.** *A triple $(\Omega, \mathcal{F}, P)$, where Ω is a sample space - set of all possible events, $\mathcal{F}$ is an event space - set of all subset of the sample space and P is a probability measure on $(\Omega, \mathcal{F})$*

For example for a coin toss it can be $\Omega = \{\text{'heads', 'tails'}\}$

Definition 2 (Discrete Random variable)**.** *A random variable X is mapping from the sample space to some measurable space E*

$$X : \Omega \to \mathbb{R} \tag{1.1}$$

If the image of the mapping is countable, it is a discrete random variable.

Definition 3 (Probability mass function)**.** *Probability mass function (or pmf) of a discrete random variable X on $(\Omega, \mathcal{F}, P)$ is the function $p_X : \mathbb{R} \to [0, 1]$ s. t.*

$$p_X(x) = P(X = x)$$

Definition 4 (Distribution function)**.** *Distribution function of a discrete random variable X on $(\Omega, \mathcal{F}, P)$ is the function $F_X : \mathbb{R} \to [0, 1]$ defined by*

$$F_X(x) = P(X \leq x)$$

It is also called a cumulative distribution function, or CDF.

Definition 5 (Expectation of a discrete random variable)**.** *If X is a discrete random variable, the expectation of X is defined by*

$$E(X) = \sum_{x \in Im(X)} xP(X = x)$$

whenever this sum converges absolutely, that is $\sum_x |xP(X = x)| < \infty$.

Definition 6 (Continuous Random variable)**.** *Random variable X is continuous, if its distribution function F_X may be written as*

$$F_X(x) = P(X \leq x) = \int_{-\infty}^{x} f_X(u)du, \ x \in \mathbb{R}$$

for some non-negative function f_X. We say that this random variable has a (probability) density function (pdf) f_X.

Definition 7 (Expectation of a continuous random variable)**.** *If X is a continuous random variable with a probability density function f_X, then the expectation of X is defined as*

$$E(X) = \int_{-\infty}^{\infty} xf(x)dx$$

Definition 8 (The chain rule (probability) [2])**.** *Let X, Y be random variables. Then*

$$P(X = x, Y = y) = P(Y = y|X = x) \cdot P(X = x)$$

For multiple random variables $X_1, ..., X_n$ taking possible values $x_1, ..., x_n$, the following holds

$$P(X = x_n, ..., X_1 = x_1) = \prod_{i=1}^{n} P(X_i = x_i|X_{i-1} = x_{i-1}, ..., X_1 = x_1)$$

Definition 9 (Bayes Theorem)**.** *Let $\{B_1, B_2, ...\}$ be a partition of the sample space Ω s.t. $\forall i : P(B_i) > 0$. For any event A with $P(A) > 0$,*

$$P(B_j|A) = \frac{P(A|B_j)P(B_j)}{\sum_i P(A|B_i)P(B_i)}$$

2. Social Networks

Social networks are an important part of our lives, and study of them is a major part of this thesis. In this chapter, we will introduce the theoretical framework used to study them, and describe properties relevant for this work. Theory included in this chapter is sourced mostly from Chapter 19 of [3] and Chapter 4 of [4].

2.1 Definitions and Properties

We will represent social networks as a graph $G = (N, E)$, where N is the set of nodes and E the set of edges. Vertices represent individuals (most of the time), and edges the relationships between them. They can be both directed or undirected, depending on the semantics. For example on Twitter, if user u follows user v, then we will have a directed edge (u, v) from u to v. If we had a network of friends for example, the edges would be undirected, since friendship is symmetrical (in most cases).

Social networks have some important properties that distinguish them from other graphs, and we will talk about them here.

2.1.1 Properties

The first one is *homophily*. It is a principle that is important for estimation of properties of nodes in social networks, and many others. It can be summed up in the adage *Birds of a feather flock together*. In other words, connected nodes are more likely to share similar properties.

Next we have something called *triadic closure*. It says that when we have two nodes with a common friend, they will likely also be connected in the future. This property is one of the reasons that social networks tend to *densify* over time. The pace of addition of edges is faster than that of nodes. Indeed, if $e(t)$ is the number of edges at time t, similarly $n(t)$ for nodes, then

$$e(t) \propto n(t)^{\beta}, 1 \leq \beta \leq 2 \tag{2.1}$$

If we have $\beta = 1$, then the average degree of nodes in the network is not affected by addition of new nodes. On the other hand, $\beta = 2$ corresponds to the network being a constant fraction of a complete graph of $n(t)$ nodes at all times.

This property is closely related to the fact that the average path distance between nodes in the network is quite small. This is called the *small world property*. The distance is thought to grow logarithmically with the number on nodes.

Related to the small average distance are *shrinking diameters*, which means, that, quite unexpectedly, the maximum length of the shortest paths between nodes is shrinking over time. This is caused by the presence of *hubs*. Hubs are highly interconnected nodes, often with degrees much higher than most other nodes in the network.

These large hubs are created in part because popular individuals tend to get more and more popular. We call this phenomenon *preferential attachment*, If the

probability of a new node in the network attaching itself to node i is $p(i)$, and the degree of node i is $d(i)$, then

$$p(i) \propto d(i)^{\alpha} \tag{2.2}$$

Parameter α is domain-dependent. If $\alpha \approx 1$, we call this network *scale-free*. We will talk about the *scale-free* property later.

As a network evolves, it tends to form a *giant connected component*, or GCC, since most new nodes will attach themselves to major hubs. GCC has to be taken into account when we are doing clustering, since it tends to make the clusters unbalanced.

2.2 Power-Law Degree distribution

A very important consequence of preferential attachment is the node degree distribution in social networks. This is the main property that distinguishes social networks from random graphs. It says that the probability p_k of a node having a degree k is proportional to

$$p_k \sim k^{-\gamma}, \gamma \geq 1 \tag{2.3}$$

Where γ is a network-dependent parameter describing the degree distribution. If we take a logarithm of the above, we realize that $\log p_k$ is linearly dependent on $\log k$. For directed graphs, we have a $p_{k_{in}}$ and $p_{k_{out}}$ for in-degrees and out-degrees. For example, the degree distribution of the World Wide Web at end of the 20th century, had a $\gamma_{in} \approx 2.1$ and $\gamma_{out} \approx 2.45$ [5].

2.2.1 Formal Definition of Power-Law Degree Distribution

A different way we can write 2.3 is

$$p_k = Ck^{-\gamma}$$

for some constant C. Since degrees can be any $k \in \mathcal{N}, k \geq 1$, and it should be a distribution, we constrain it by

$$\sum_{k=1}^{\infty} p_k = 1$$

Combining these two formulas we can derive the value for C

$$C \sum_{k=1}^{\infty} p_k = 1$$

$$C = \frac{1}{\sum_{k=1}^{\infty} k^{-\gamma}} = \frac{1}{\zeta(\gamma)} \tag{2.4}$$

where $\zeta(\gamma)$ is the Riemann Zeta function. So the probability of a node having a degree k is

$$p_k = \frac{k^{-\gamma}}{\zeta(\gamma)} \tag{2.5}$$

However, sometimes it is convenient to allow values of node degrees to be any positive real number. We will treat the probability p_k as a function of k. Assuming k_{min} is the minimum degree of a node in the network, we write

$$p(k) = Ck^{-\gamma} \tag{2.6}$$

Once again, for this to be a distribution, the area under the curve has to sum up to 1, so

$$\int_{k_{min}}^{\infty} p(k)dk = 1$$

Similarly as before, we arrive at

$$C = \frac{1}{\int_{k_{min}}^{\infty} k^{-\gamma}dk} = (\gamma - 1)k_{min}^{\gamma-1} \tag{2.7}$$

$$p(k) = (\gamma - 1)k_{min}^{\gamma-1}k^{-\gamma} \tag{2.8}$$

2.2.2 Effects of Power-law distribution

As we said before, networks following the power law distribution differ from random networks mainly by the presence of *hubs*. In random networks, the degrees of nodes follow the Poisson distribution, and therefore cluster around the mean, denoted by $\langle k \rangle$. In scale free networks, there is a nontrivial amount of very high degree nodes. This would be very improbable in random networks. Scale-free networks also allow for very high maximum degrees, orders of magnitudes higher than k_{min}.

For illustration, in a Poisson distribution, the probability of a node having a degree of 100 hundred is

$$p_{100} \approx 10^{-94}$$

which is effectively zero. However, for a power-law, $p_{100} \approx 4 \cdot 10^{-4}$. If we had a random network with $\langle k \rangle = 4.6$ and $N = 10^{12}$ (Similar to WWW), then the expected number of nodes with a degree higher than 100 would be

$$N_{k \geq 100} = 10^{12} \sum_{k=100}^{\infty} \frac{(4.6)^k}{k!} e^{-4.6} \approx 10^{-82}$$

Which is effectively zero. Naturally, there exist many web pages with more than 100 other documents linking to them, so the Poisson distribution is not a good model for our network. However, if it followed a power-law with $\gamma_{in} = 2.1$ (as estimated for WWW), we would expect to see

$$N_{k \geq 100} = 4 \cdot 10^9$$

nodes with a degree higher than 100. This is a very large number, but given the interconnectedness of real networks, and the fact that our network has a trillion nodes, it is not a bad estimate.

2.2.3 Large Hubs

Why does the power-law distribution allow for a high maximum degree k_{max}? It is better to illustrate this using the exponential distribution.

$$p(k) = Ce^{\lambda k} \tag{2.9}$$

where λ is the mean of the distribution. From the normalization condition we get the value for C

$$\int_{k_{min}}^{\infty} p(k)dk = 1$$

$$C = \lambda e^{\lambda k_{min}}$$

To calculate k_{max}, we assume that we expect at most 1 node in the regime (k_{max}, ∞). This means that the probability of finding a node whose degree is $\geq k_{max}$ is $\frac{1}{N}$.

$$\int_{k_{max}}^{\infty} p(k)dk = \frac{1}{n}$$

$$k_{max} = k_{min} + \frac{lnN}{\lambda} \tag{2.10}$$

We see that for the exponential distribution, maximum degree scales logarithmically with the number of nodes in the network.

In case of the power-law distribution, using the same reasoning and the formula 2.8, we get

$$k_{max} = k_{min} N^{\frac{1}{\gamma - 1}} \tag{2.11}$$

This scales much faster than $\log N$, and implies that large networks can have huge k_{max}, sometimes called *natural cutoffs*.

2.3 Fitting and generating power-laws

Part of this thesis is to estimate the degree exponent γ of the studied networks. To find the exponent of a given degree distribution, we will use the method introduced in [6].

For the discrete power-law, we can use the *Maximum likelihood estimate* (MLE) for γ from [7][8]. Assuming our network contains N nodes, and $k_{min} = 1$, the MLE estimate is

$$\frac{\zeta'(\hat{\gamma})}{\zeta(\hat{\gamma})} = -\frac{1}{N}\sum_{i=1}^{n} \ln k_i \tag{2.12}$$

Otherwise, the estimate is [9] [10]:

$$\frac{\zeta'(\hat{\gamma}, k_{min})}{\zeta(\hat{\gamma}, k_{min})} = -\frac{1}{N}\sum_{i=1}^{n} \ln k_i \tag{2.13}$$

Where $\zeta(\hat{\gamma}, k_{min})$ is the Hurwitz zeta function, which is a generalized Riemann zeta defined as

$$\zeta(s, a) = \sum_{n=0}^{\infty} \frac{1}{(n+a)^s} \tag{2.14}$$

Equivalent to this is a direct numerical maximization of the logarithm of the likelihood function, which is

$$\mathcal{L}(\alpha) = -n \ln \zeta(\alpha, x_{min}) - \alpha \sum_{i=1}^{n} \ln x_i \tag{2.15}$$

This is the method which we are going to use.

However, using the real k_{min} might not result in a good fit for a power-law. The degree distribution of a real network often does not scale according to a power law at the beginning. It only starts to follow it after some cutoff degree k. To find such k, serving as a k_{min}, we will scan all $k \in [k_{min}, k_{max}]$. For each k, we will calculate $\hat{\gamma}$ using previously described MLE. Then, we will calculate the Kolmogorov-Smirnov (KS) statistic for each one. This constitutes determining the maximum distance D between the CDF of the empirical distribution of the data $S(k)$, and the power-law CDF parametrized by our estimate $\hat{\gamma}$

$$D = \max_{k \geq k_{min}} |S(k) - \left[1 - \frac{\zeta(\hat{\gamma}, k)}{\zeta(\hat{\gamma}, k_{min})}\right]$$

For a given k_{min}, we choose the $\hat{\gamma}$ which minimizes D. At some point, the number of nodes whose degree is higher than the k_{min} candidate we are trying right now may be too low to get a statistically significant estimate of γ for the whole network. Thus, we will stop the search when the number of nodes with a degree $d \geq k$ is smaller than a number of our choosing.

To get standard errors for the estimates, we can sample N observations uniformly from the original data, calculate estimates of parameters for this sample, and compute the averages and standard errors from multiple samplings. This is the "bootstrap" method described in [11].

2.3.1 Goodness-of-fit

We will use a goodness-of-fit test to determine how good does the power-law with estimated parameters fit the observed data. Our hypothesis is that the observed data comes from a power-law distribution with the previously estimated parameters.

To express confidence in our hypothesis in a numerical way, we will use a p-value. If this p-value is small, smaller than our chosen significance level, then we rule out our power-law hypothesis. We will use the same significance level as in [6], that is $p \leq 0.1$.

To calculate this p-value, we will generate synthetic power-law distributed datasets, with the same parameters as the ones we have estimated. Then, we fit these datasets to their own power laws, and calculate their KS statistics relative to their own best-fit power laws. The fraction of the datasets with a KS statistic larger than the KS statistic of our own model for the empirical data will be the p-value. We can interpret p as follows: With probability p or less we would only by chance observe data that fits as poorly to the model as the data we have.

However, when generating these datasets, we want their distributions to follow the power-law in the regime $[k_{min}, \infty)$, but to be similar to our observations below it. Let n be the total number of nodes observed and n_{tail} the number of observed

nodes with degrees greater than k_{min}. Then, with probability n/n_{tail} we sample from a power law with parameters $\hat{\gamma}$ and k_{min}. Otherwise, we sample degree values from nodes in our original data, whose degree is less than or equal to k_{min}, uniformly randomly.

The accuracy of our p estimate depends on the number of synthetic datasets we generate. In [6], *Clauset et al.* recommend generating at least $\frac{1}{4}\varepsilon^{-2}$ datasets, if we want our p estimate to be accurate within ε of the true value.

2.3.2 Generating power-laws

To sample from a power-law distribution with given γ and k_{min}, we will use the method from [6]. Usually, we have a random number generator, which can generate uniformly distributed real numbers $r \in [0, 1]$. The corresponding probability density will be $p(r)$. We want to transform this r into a number sampled from a different, arbitrary distribution, whose density we will call $p(x)$. The relation between these densities is described by this formula

$$p(x) = p(r)\frac{\mathrm{d}r}{\mathrm{d}x} = \frac{\mathrm{d}r}{\mathrm{d}x}$$

Integrating w.r.t x we get

$$P(x) = \int_x^\infty p(x')dx' = \int_1^r dr' = 1 - r \tag{2.16}$$

therefore

$$x = P^{-1}(1 - r)$$

Where P^{-1} is the functional inverse of the CDF of the other distribution. We now set P to be a power-law distribution from which we want to generate samples. For a continuous power law we get

$$x = k_{min}(1 - r)^{\frac{-1}{\gamma-1}} \tag{2.17}$$

An equivalent of 2.16 for a discrete power law is

$$P(x) = \sum_{x'=x}^{\infty} p(x') = 1 - r$$

However, in the discrete case we cannot directly write an expression for P^{-1}. We have to solve the expression numerically.

2.4 Important nodes

There are many metrics by which we can measure the importance of nodes in a social network, and here we will mention the ones used in this thesis.

2.4.1 Betweennes Centrality

This metric assigns importance to a node based on the flow of information through it. It is based on the number of shortest paths that pass through this node. Let

$r \leftarrow$ Uniformly randomly generated number, $0 \leq r \leq 1$;
$x_2 \leftarrow x_{min}$;
repeat
  $x_1 \leftarrow x_2$;
  $x_2 \leftarrow 2x_1$;
until $P(x_2) < 1 - r$;
Determine k s.t. $k \leq x < k + 1$ using binary search, and discard the non-integer part of k;
return *Power-law distributed integer k*

Algorithm 1: Generate power-law sample

q_{jk} be the number of shortest paths between nodes j and k. Let $q_{jk}(i)$ be the number of paths between these nodes passing through node i. Then the fraction of shortest paths between j and k that pass through i is

$$f_{jk}(i) = \frac{q_{jk}(i)}{q_{jk}}$$

Betweennes centrality is then defined as

$$C_B(i) = \frac{\sum_{j<k} f_{jk}(i)}{\binom{n}{2}}, C_B \in [0, 1]$$

The denominator represents the number of all pairs of nodes.

2.4.2 PageRank

Contents of this and the following section are sourced in Chapter 18.4 of [3].

PageRank is a method that models the importance of nodes based on random walks in the network. It was famously invented by the founders of Google, Larry Page and Sergey Brin, and was used by Google to rank pages in their search results. However, it can be used as an importance measure in social networks as well.

We will consider a surfer model, that starts at a random node and moves uniformly randomly between connected nodes. We want to know the long term probability of the surfer finding himself at any one node, which is called the *steady state probability*.

However, some nodes may not have any outgoing connections, we will call them *dead ends*. For these nodes PageRank cannot be defined. Alternatively there can even be groups of nodes, such that when the surfer arrives in one of them, he cannot move out of the group. These are called *absorbing components*. Another problem could be that the graph is disconnected. In this case, we wouldn't be able to compare nodes within different components properly.

The solution in the PageRank method for the first problem is to add links to every other node in the network, and to itself, with a transition probability $1/n$. For the other two problems, we will use *teleportation*. The surfer may at each step decide to visit any other node in the network with probability α, or follow the links from this node with probability $(1 - \alpha)$. Typically, $\alpha = 0.1$. This can

be thought as a kind of *smoothing*, since higher values of α will make the steady state distribution more even.

Computing the steady state distribution proceeds as follows. Let $In(i)$ be the set of nodes incident to i, and similar $Out(i)$ set of the other ends of outgoing edges of i. We will treat the network as a *Markov chain* described by a transition matrix $P \in \mathbb{R}^{n \times n}$, where n is the number of nodes. Probability of transitioning from node i to j is

$$p_{ij} = \frac{1}{|Out(i)|} \tag{2.18}$$

Probability of teleportation from some node into node i is

$$p_{teleport}(i) = \frac{\alpha}{n} \tag{2.19}$$

since we can teleport into it from anywhere (and from itself too). If we denote the steady-state probability of node j as $\pi(j)$, the probability of transitioning into i is

$$p_{trans}(i) = (1 - \alpha) \cdot \sum_{j \in In(i)} \pi(j) \cdot p_{ji} \tag{2.20}$$

The steady-state probability of node i will be the sum of the former probabilities

$$\pi(i) = \alpha/n + (1 - \alpha) \cdot \sum_{j \in In(i)} \pi(j) p_{ji} \tag{2.21}$$

In matrix form, for all nodes

$$\boldsymbol{\pi} = \alpha \frac{\mathbf{e}}{n} + (1 - \alpha) \mathbf{P}^T \boldsymbol{\pi} \tag{2.22}$$

where $\mathbf{e}$ is a column vector of all ones. The steady state probabilities for all nodes have to sum to one, because it is a distribution.

$$\sum_{i=1}^{n} \pi(i) = 1 \tag{2.23}$$

This system of linear equations is solvable using the following iterative method. First, we initialize $\pi^{(0)}(i) = 1/n, \forall i$, and then we repeat

$$\boldsymbol{\pi}^{(t+1)} \leftarrow \alpha \frac{\mathbf{e}}{n} + (1 - \alpha) \mathbf{P}^T \boldsymbol{\pi}^{(t)} \tag{2.24}$$

We continue until the distribution change compared to the previous iteration is less than a user-defined threshold ϵ.

Input: Transition matrix $\mathbf{P}$, smoothing parameter α, threshold ϵ
$\boldsymbol{\pi}^{(0)} \leftarrow \mathbf{e}/n$;
repeat
  $\boldsymbol{\pi}^{(t+1)} \leftarrow \alpha \frac{\mathbf{e}}{n} + (1 - \alpha) \mathbf{P}^T \boldsymbol{\pi}^{(t)}$;
  Scale $\boldsymbol{\pi}$ so that $\sum_i \pi(i) = 1$;
until $\boldsymbol{\pi}^{(t+1)} - \boldsymbol{\pi}^{(t)} < \epsilon$;
return *PageRank vector* $\boldsymbol{\pi}$

Algorithm 2: Power Iteration method

2.4.3 HITS

Hypertext Induced Topic Search, or HITS, is another algorithm used to rank the importance of pages in search results. It scores the nodes as *authorities* and *hubs*. Authorities are pages which contain many in-links, and hubs are pages containing links to many authorities. The assumption is that good authorities are pointed to by good hubs. In-turn, good hubs contain many links to good authorities. Each page is assigned both an *authority score* $a(i)$ and a hub score $h(i)$. We want their L_2 norms of both vectors of scores to be 1, i.e.

$$\sqrt{\sum_{i=1}^{n} a(i)^2} = \sqrt{\sum_{i=1}^{n} h(i)^2} = 1 \tag{2.25}$$

We can apply this algorithm to social networks too, since nodes with a high authority or hub score might be very influential in the network.

Similar to PageRank, we iteratively update $a(i)$ and $h(i)$, until their values converge. We define $a(i)$ and $h(i)$ as follows

$$h(i) = \sum_{j \in Out(i)} a(j) \tag{2.26}$$

$$a(i) = \sum_{j \in In(i)} h(j) \tag{2.27}$$

The algorithm first sets the $h^{(0)} = a^{(0)} = 1/\sqrt{n}$, and each iteration updates the vectors of scores.

Input: Adjacency matrix $\mathbf{A}$
$\mathbf{h}^{(0)} = \mathbf{a}^{(0)} \leftarrow \mathbf{e}/\sqrt{(n)}$;
repeat
  $\mathbf{a}^{(t+1)} = \mathbf{A}^T\mathbf{h}^{(t)}$;
  $\mathbf{h}^{(t+1)} = \mathbf{A}\mathbf{a}^{(t)}$;
  Normalize $\mathbf{a}^{(t+1)}$ and $\mathbf{h}^{(t+1)}$ so that their L_2-norms are both equal to 1;
until $\boldsymbol{h}^{(t)}$ *and* $\boldsymbol{a}^{(t)}$ *converge*;
return *authority score vector* $\boldsymbol{a}$*, hub score vector* $\boldsymbol{h}$

Algorithm 3: HITS algorithm

3. Models

This chapter includes the theoretical description of the models studied and used for classification tasks solved in the thesis.

3.1 Naive Bayes classifier

This probabilistic model is the simplest one out of the models we have tested, and yet it has shown promising results. It is based on the Bayes' Theorem, which is a basic result from probability theory. Theory in this section is adapted from Chapter 8 of [12], unless stated otherwise.

Let us assume we want to know how likely is the result X that we have observed, given prior data Y. If X can attain values $X_1, X_2, .., X_k$, then the probability of observing some X_i given the data Y is

$$P(X_i|Y) = \frac{P(Y|X_i)P(X_i)}{\sum_{j=1}^{k} P(Y|X_j)P(X_j)} \tag{3.1}$$

In this case we call $P(X_i|Y)$ the *posterior*, which represents the distribution of the X given the data Y, and $P(X_i)$ the *prior*, which represents the probability of observing the value X_i independent of any other data. The term $P(Y|X_i)$ is called a *likelihood*, because it says how likely is the data given the result, and it is a function of the data. The denominator is derived using the law of total probability (or partition theorem) [13], which is for the discrete case as follows

$$P(Y) = \sum_{i=1}^{n} P(Y|X_i)P(X_i) \tag{3.2}$$

Now let us assume that our data consists of number of observations, each being a D-dimensional vector of features $\mathbf{x} = (x_1, x_2, ..., x_D)$. Each observation belongs to exactly one of k classes $C_1, ..., C_k$. Assuming we already have a prior over all classes, the probability of $\mathbf{x}$ belonging to class C_i is as follows:

$$P(C_i|\mathbf{x}) = \frac{P(\mathbf{x}|C_i)P(C_i)}{P(\mathbf{x})} \tag{3.3}$$

The posterior might be hard to model outright.
This is because $P(\mathbf{x}|C_i)P(C_i) = P(x_1, x_2, ..., x_D, C_i)$ via the chain rule. Then

$$\begin{aligned} P(\mathbf{x}|C_i)P(C_i) &= P(x_1, x_2, ..., x_D, C_i) \\ &= P(x_1|x_2, x_3, ..., C_i)P(x_2, x_3, ..., x_D, C_i) \\ &= P(x_1|x_2, x_3, ..., C_i)P(x_2|x_3, ..., x_D, C_i)P(x_3, ..., x_D, C_i) \\ &= P(x_1|x_2, x_3, ..., C_i)P(x_2|x_3, ..., x_D, C_i)...P(x_D, C_i)P(C_i) \end{aligned}$$

Instead of modelling all of these probabilities, we can use the *naive* assumption. It says that, given the class C_i, all the features are independent. It is a very strong assumption, which might not necessarily reflect the real data. However, it

makes the probabilities a lot easier to model. So now we can rewrite the above probability to this form

$$P(\mathbf{x}|C_i) = P(x_1, x_2, ..., x_D, C_i) = P(x_1|C_i)P(x_2|C_i)...P(x_D|C_i) = \prod_{j=1}^{D} P(x_j|C_i) \tag{3.4}$$

Substituting to 3.3 we get

$$P(C_i|\mathbf{x}) = \frac{\prod_{j=1}^{D} P(x_j|C_i)P(C_i)}{P(\mathbf{x})} \tag{3.5}$$

Now we use the following rule to predict the classes of each data point:

$$\begin{aligned} \arg\max_k p(C_k|\mathbf{x}) &= \arg\max_k \frac{\prod_{j=1}^{D} P(x_j|C_k)P(C_k)}{P(\mathbf{x})} \\ &= \arg\max_k \prod_{j=1}^{D} P(x_j|C_k)P(C_k) \end{aligned}$$

Since $P(\mathbf{x})$ does not depend on the choice of C_k, we can omit it from the formula. This rule is called the *Maximum a posteriori* estimation, or MAP. In general, let us say we have a model with parameters $\mathbf{w}$(also called *weights*) that we want to train, and we also have data $\mathbf{X}$. Then the *Maximum a posteriori estimate* for the parameters $\mathbf{w}$ is

$$\mathbf{w} = \arg\max_{\mathbf{w}} P(\mathbf{X}|\mathbf{w})P(\mathbf{w}) \tag{3.6}$$

There is a technique called *Maximum likelihood estimation*, which is similar to MAP, but we try to find parameters such that we maximize the likelihood of the data. In other words

$$\mathbf{w} = \arg\max_{\mathbf{w}} P(\mathbf{X}|\mathbf{w}) \tag{3.7}$$

We can see that MLE is similar to MAP, we just don't use any prior in the former.

What we need to do now is to model the prior probabilities of classes $P(C_i)$, and the likelihoods $P(x_j|C_i)$. To calculate a prior, a reasonable thing is to just look at our data and assign to each class its frequency. So if k_i out of N observations belong to class C_i, its probability is

$$P(C_i) = \frac{k_i}{N} \tag{3.8}$$

Now, based on how we model the probabilities $P(x_j|C_i)$, we have different types of Naive Bayes classifiers.

3.1.1 Gaussian Naive Bayes

In this variant, we assume that the features are being drawn from a Normal distribution. We model the individual probabilities as $P(x_j|C_i) \sim N(\mu_{j,i}, \sigma^2_{j,i})$, that is, each feature is assumed to have been drawn from a Normal distribution with a mean of $\mu_{j,i}$ and a variance of $\sigma^2_{j,i}$. Assuming we have observations

$x_1, x_2, ..., x_{N_k}$ for class k, and each of them consists of D features, using MLE we obtain estimates for the means as

$$\mu_{d,k} = \frac{1}{N_k} \sum_{i=1}^{N_k} x_{i,d} \tag{3.9}$$

where N_k is the number of documents of class k, each $x_i = (x_{i,1}, x_{i,2}, ..., x_{i,D})$, and we are looking for the mean of the d-th feature. The variations will then be

$$\sigma_{d,k} = \frac{1}{N_k} \sum_{i=1}^{N_k} (x_{i,d} - \mu_{d,k})^2 \tag{3.10}$$

We can also add some constant α to all of them, so that the distributions are a bit wider.

3.1.2 Multinomial Naive Bayes

We can also model the feature probabilities using the Multinomial distribution. Assuming we have a vector of non-negative integer values $\mathbf{x} = (x_1, ..., x_D)$, and we have observed the outcome of n trials, the probability of seeing the values $\mathbf{x}$ is

$$P(\mathbf{x}) = \binom{n}{x_1, ..., x_D} p_1^{x_1} p_2^{x_2} ... p_D^{x_D} \tag{3.11}$$

We can see that this distribution is parametrized by the distribution $\mathbf{p}$ of probabilities p_i, which sum to 1. $\binom{n}{x_1,...,x_D}$ is the Multinomial coefficient, which is equal to $\frac{n!}{x_1! x_2! ... x_D!}$. So our parameters will be probabilities $p_{d,k}$. Using MLE we get that

$$p_{d,k} = \frac{n_{d,k}}{\sum_j n_{j,k}} \tag{3.12}$$

where $n_{j,k}$ is the sum of features x_d for a given class C_k. As with the Bernoulli Naive Bayes, we can use Laplace smoothing to add small factor α to all feature sums:

$$p_{d,k} = \frac{n_{d,k} + \alpha}{\sum_j (n_{j,k} + \alpha)} \tag{3.13}$$

3.2 Simple Neural Networks

Neural networks are a broad family of machine learning models, which have become quite popular in today's machine learning research and practical application. In this section, we are going to talk about how a simple neural network called the Multi-layer Perceptron.

3.2.1 Multilayer Perceptron

This section includes information mostly from Chapter 5 of [12]. Multilayer perceptron (MLP) consist of multiple layers of *perceptrons*, which are modeled after real neurons. We can visualize the topology of a network as a directed acyclic graph, where nodes are the perceptrons and edges represent incoming and

outgoing connections between them. The network consists of multiple layers. There are no connections between nodes inside a layer.

First we will have an *input* layer, with nodes for each input dimension labeled $x_1, ..., x_D$. These nodes have no inputs, and their output is just the value of the input vector at a given index.

We will also have an output node for every output dimension that we want. So if we want to classify data as one of K classes, we will have K output nodes $y_1, ..., y_K$. This is called the *output* layer.

Finally, in-between these layers we will have an arbitrary number (> 0) of so-called *hidden* layers, which will contain regular nodes. Every node has a link to every node in the previous layer. It also has an edge leading to every node in the next layer.

You can see an example MLP with one hidden layer in Figure 3.2.1.

Every neuron contains an *activation function*, which computes its output based on the weighted sum of the outputs of incident neurons.

$$a(y(\mathbf{x}; \mathbf{w})) = a\left(\sum_{i=1}^{n} x_i w_i + b\right) \tag{3.14}$$

Here x_i is the output of i-th neuron in the previous layer, and w_i is the corresponding weight. We also add a bias to the whole sum, which is also a trainable parameter, together with weights $\mathbf{w}$.

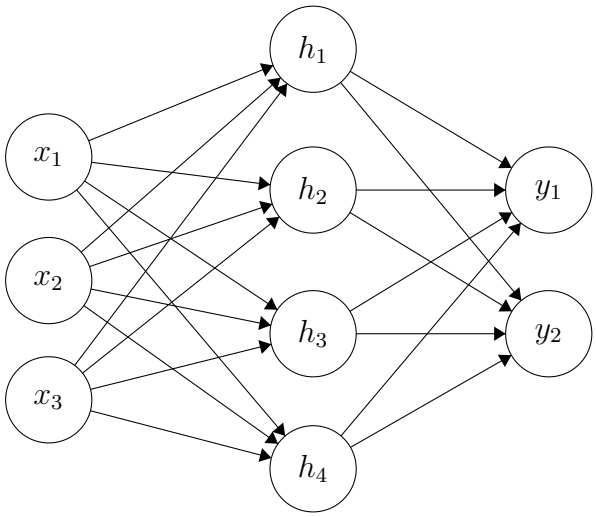


Figure 3.1: MLP Example

Since each hidden and output neuron has its own weights for all the different inputs, we can store all the weights of one layer in a matrix $\mathbf{W} \in \mathbb{R}^{N \times M}$, where N is the number of nodes in this layer, and M is the number of nodes in the previous layer. Similarly, we can store all the biases of nodes in a single layer in one vector $\mathbf{b} \in \mathbb{R}^N$. Different hidden layers may have different activation functions. If we denote $h_i^{(k)}$ the output value of i-th node in the k-th layer, similarly for weights $\mathbf{w}^{(k)}$ and biases $\mathbf{b}^{(k)}$, the output values of each hidden node in layer k can be calculated as

$$h_i^{(k)} = a\left(\sum_{i=1}^{N} h_i^{(k-1)} w_{i,j}^k + \mathbf{b}^{(k)}\right) = a\left(\mathbf{h}^{(k-1)} W^{(k)} + \mathbf{b}^{(k)}\right) \tag{3.15}$$

Popular activation functions for the hidden layer include

- $ReLU(x) = \max(0, x)$
- sigmoid - $\sigma(x) = \frac{1}{1+e^{-x}}$
- $tanh(x) = 2\sigma(2x) - 1$
  - symmetrizes the sigmoid and makes the derivative in 0 equal to 1

Based on our task, we choose different output layer activations

- identity - we want to perform linear regression (output a numerical value)
- $\sigma(x)$ - we perform binary classification
- softmax - K-class classification, outputs a probability distribution
  - $softmax(x)_i = \frac{e_i^x}{\sum_j^K e^{x_j}}$

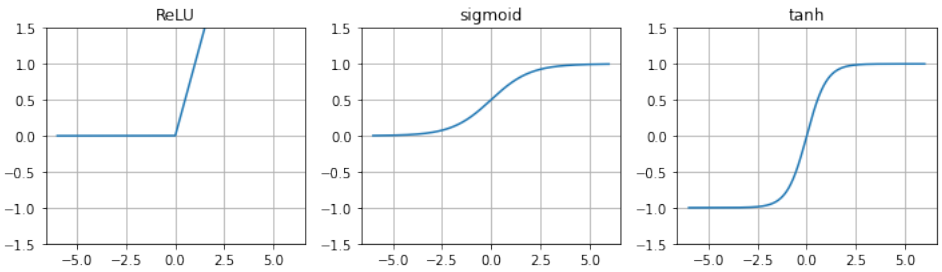


Figure 3.2: Activation function graphs

It can be proven, that given a monotonically increasing, sigmoidal activation function, MLP with just 1 hidden layer (with a finite number of neurons) can approximate any continuous function, to an arbitrary precision [14].

Our goal during training is to find suitable weights and biases for all layers. We do this by minimizing an error function. This function $E(\mathbf{w})$ tells us the difference between the target values and the predictions of our model, with the current weights $\mathbf{w}$. A notable error function is called *mean square error*, or MSE for short. It is computed as follows:

$$MSE(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(y(\mathbf{x}_i; \mathbf{w}) - t_i)^2 \tag{3.16}$$

where N is the total number of training examples, $y(\mathbf{x}_i; \mathbf{w})$ the actual output of our model for the i-th vector, and t_i its target value. We will refer to any error function we attempt to minimize as a *loss*. We can also add an L_2 regularization term with parameter λ, which penalizes large weights, since in practice it was shown that smaller weights lead to better results. So the modified loss will be

$$MSE(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(y(\mathbf{x}_i; \mathbf{w}) - t_i)^2 + \frac{\lambda}{2}||\mathbf{w}||^2 \tag{3.17}$$

Parameter λ is preset by the user, so we call it a *hyperparameter*. We train the MLP using an algorithm called *Stochastic Gradient Descent* (SGD). SGD iteratively updates the weights at each step, minimizing the loss in the direction of the steepest descent. It converges almost surely to a global minimum of the loss function, provided it is convex, otherwise it converges to a local one. An outline is given below

Input: Training Dataset ($\mathbf{X} \in \mathbb{R}^{N \times D}, \mathbf{t} \in \{-1, 1\}^N$
`// all weight matrices are represented by w`
initialize $\mathbf{w}$ randomly ;
set biases to 0 ;
repeat
  `// Process a minibatch of examples of size b`
  `//` $E_i(\mathbf{w})$ `is the loss w.r.t the` i`-th training sample`
  $\mathbf{g} \leftarrow \frac{1}{|\mathbf{b}|} \sum_{i \in \mathbf{b}} \nabla_{\mathbf{w}} E_i(\mathbf{w})$;
  $\mathbf{w} \leftarrow w - \alpha \mathbf{g}$;
until *convergence or maximum number of iterations is reached*;
return $\boldsymbol{w}_t$

Algorithm 4: Minibatch SGD

Once again we have a hyperparameter $\alpha, \alpha \in [0, 1]$, which we call the *learning rate*. It tells us how much the computed gradient shifts the weights in each step. However, better performing ways of training MLPs exist, namely an algorithm called Adam. We will be using Adam to train these and other types of networks in the experimental part of the thesis.

Adam is an improvement over SGD, in the sense that instead of a single unchanging learning rate, it maintains separate learning rates for each network parameter, and updates them during training. Specifics can be found in this paper [15]. Learning rates are changed based on the so-called first and second order moments of the gradients. It uses exponential moving average (EMA) to smooth the moments. We can control the decay of these EMAs with parameters β_1 and β_2 for the first and second order moment respectively. The EMA for a series X is defined recursively:

$$S_t = \begin{cases} X_0 & t = 0 \\ S_{t-1} & \text{otherwise} \end{cases}$$

Where S_t is the value of EMA at time t and X_t is the value at time t. Here is the outline of the Adam algorithm paraphrased from [15]

The ε is a small constant $\sim 10^{-8}$ to avoid problems with division by zero. Good hyperparameter choices for most ML applications are $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$.

Input: Parameter vector $\mathbf{w}$
Input: learning rate α
Input: Exponential decay rates $\beta_1, \beta_2 \in [0, 1)$
Input: Stochastic objective function $f(\mathbf{w})$
$\mathbf{m}_0 \leftarrow 0$; `// Initialize 1st moment vector`
$\mathbf{v}_0 \leftarrow 0$; `// Initialize 2nd moment vector`
$t = 0$; `// Initialize time-step`
while $\boldsymbol{w}$ *not converged* **do**
  $t \leftarrow t + 1$;
  `// Get Gradients w.r.t stochastic objective at time` t
  $\mathbf{g}_t \leftarrow \nabla_{\mathbf{w}} f_t(\mathbf{w}_{t-1})$;
  `// Update biased first moment estimate`
  $\mathbf{m}_t \leftarrow \beta_1 \cdot \mathbf{m}_{t-1} + (1 - \beta_1) \cdot \mathbf{g}_t$;
  `// Update biased second raw moment estimate`
  $\mathbf{v}_t \leftarrow \beta_2 \cdot \mathbf{v}_{t-1} + (1 - \beta_2) \cdot \mathbf{g}_t^2$;
  `// Compute bias-corrected first moment estimate`
  $\hat{\mathbf{m}}_t \leftarrow \mathbf{m}_t / (1 - \beta_1^t)$;
  `// Compute bias-corrected second moment estimate`
  $\hat{\mathbf{v}}_t \leftarrow \mathbf{v}_t / (1 - \beta_2^t)$;
  `// Update parameters`
  $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \alpha \cdot \hat{\mathbf{m}}_t / (\sqrt{\hat{\mathbf{v}}_t} + \varepsilon)$;
end
return $\boldsymbol{w}_t$

Algorithm 5: Adam optimization algorithm [15]

3.3 Recurrent Neural Networks

Recurrent Neural Networks, or RNNs for short, are a type of neural network that are made up of cells, which contain a connection back into themselves. They are best suited for data made out of sequences. This connection into itself represents the information flow during the processing of the input. In reality, there is a cell that processes information from each step of the sequence. Information from a cell for step $t - 1$ is routed to cell at step t and so on. The basic diagram is in Figure 3.3

We assume that the input is a series of vectors $\mathbf{x}^{(t)}$, and for a cell at step t, we denote its *hidden* state by $\mathbf{h}^{(t)}$ and output by $\mathbf{y}^{(t)}$. The cells usually combine their input with the state of the previous step, and output their new state forward. In the past, simple cells have been proposed, but they had a problem of vanishing(exploding) gradients.

The problem was that the state change boiled down to a repeated application of the same function. This was akin to multiplication of the same matrix as many times as there were time-steps, so for long sequences, the internal state vectors reduced to zero or exploded in size.

A few approached have been successful at dealing with this problem, namely LSTM and later GRU.

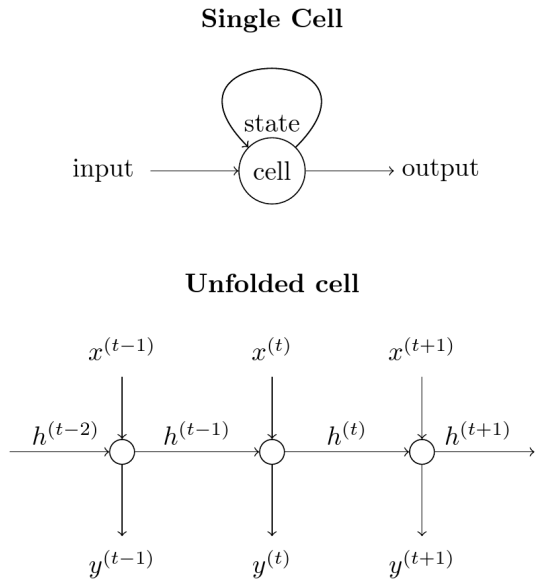


Figure 3.3: RNN cell diagram

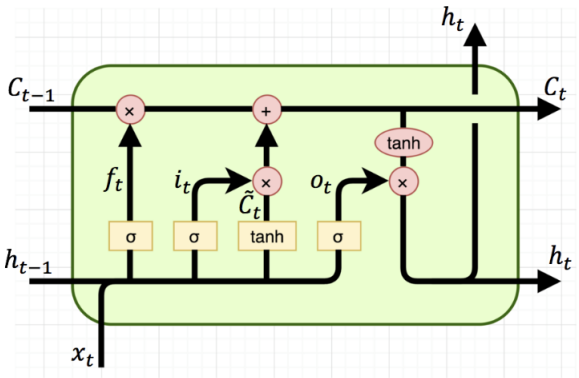


Figure 3.4: LSTM Cell detail, from Kaushik Mani's article [18]

3.3.1 LSTM

LSTM, first defined in [16], stands for *Long Short-term Memory*, and is a type of RNN cell with an internal state and importantly an internal memory. In both this subsection and subsection 3.3.2 we will use theory from [17]. An important feature of the memory is that it stores information about the sequence, but it can also *forget* it.

The LSTM cell outputs it hidden state. The internal memory can be thought of as another state, here referred to as a *cell* state C_t.

As we can see in Figure 3.4, the cell is actually made up of several Neural Network (NN) layers, called *gates*. Each of them have their own weight matrix $\mathbf{W}$, and a different activation function based on their purpose.

The calculation of a new cell state C_t and hidden state h_t is done sequentially.

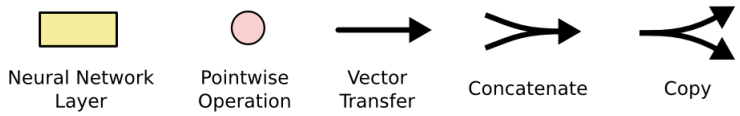


Figure 3.5: LSTM Cell legend, from C. Olah's blog [19]

Every step corresponds to a calculation involving one of the gates:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) \tag{3.18}$$
$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \tag{3.19}$$
$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_C[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_C) \tag{3.20}$$
$$\mathbf{C}_t = \mathbf{f}_t * \mathbf{C}_{t-1} + \mathbf{i}_t * \mathbf{C}_T \tag{3.21}$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_o[h_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \tag{3.22}$$
$$\mathbf{h}_t = \mathbf{o}_t * \tanh(\mathbf{C}_t) \tag{3.23}$$

Where $*$ is vector multiplication, and $[x, y]$ is vector concatenation.

First we have the forget gate f_t. This part of the network decides what information to drop from the memory cell, based on the previous hidden state h_{t-1} and current input value x_t. This is done by multiplying the old cell state by a vector of numbers between 0 and 1, which is the output of this gate because of its sigmoid activation.

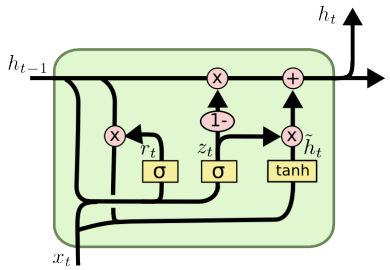
Then we want to update the memory cell with new information. This is done by the *input* gate i_t, which consists of a sigmoid NN layer and *tanh* layer. The sigmoid layer i_t decides which parts of the memory cell will be updated, and the *tanh* layer $\tilde{C}_t$ gives us a vector of candidates for the update. We multiply the candidate vector with the sigmoid vector, which has exactly the effect of choosing the candidates. We add this result to the memory cell vector, which, by now, has been modified by the forget gate.

Finally, we use the *output* gate to decide what we want as a new hidden state h_t, and therefore also the output. It will be the new cell state modified by a sigmoid based on the previous hidden state and the output. Before we output it, we apply *tanh* to it to push the values between -1 and 1.

3.3.2 GRU

GRU [20], *Gated Recurrent Unit*, is a modification of the LSTM cell, which has no memory cell and couples the forgetting and updating of the hidden state together. Diagram can be found in Figure 3.6. The main benefit of this cell is that it is simpler in composition and has a similar performance [21] in many applications. This coupling is equivalent to setting $f_t = 1 - i_t$.

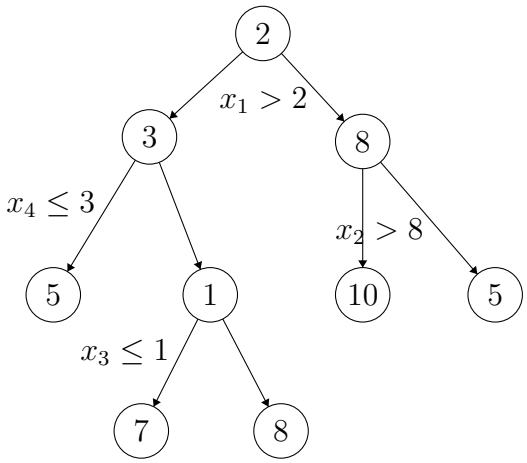
Here we have a relevance gate r_t, which has the combined functions of the forget and input gates. Another change is the update gate z_t which decides what information do we retain from the previous hidden state.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 3.6: GRU cell diagram with equations, from C. Olah's blog [22]

Figure 3.7: Decision tree example



3.4 Decision Trees

Decision trees are simple ML models, which try to partition the input space into cuboid regions in a way that maximizes homogeneity inside these regions. We will be talking about *Classification and regression trees*[23] or CART, and the theory will be from Chapter 14 of [12] unless stated otherwise.

Decision trees are modeled as binary trees, and during classification or regression, we traverse the tree starting from the root, and make decisions based on the conditions inside inner nodes. When we reach a leaf, it will contain a number or a label which will be our result. Inner nodes define the input space partitioning by choosing an input feature and conditioning it based on some numerical value. After we have the space partitioned, we can use a simpler model to solve the problem inside each region.

In general, our trees will be made up of nodes $\mathcal{T}$. Each inner node will contain a feature index $i_{\mathcal{T}}$, and a threshold value $\theta_{\mathcal{T}}$. If the value of the $i_{\mathcal{T}}$-th feature of the input vector is $> \theta_{\mathcal{T}}$, we move to the right, otherwise we move to the left. Each leaf corresponds to some region in input space, bounded by the thresholds in the inner nodes.

For example, let us say we have the tree in Figure 3.7, and we want to get an output number for $\mathbf{x} = (-1, 7, 2, 3)$. We start at the root and check if $x_1 > 2$. It is not, so we continue to the left child. Now we check if $x_4 > 3$, and so on. In the

end we reach a leaf containing number 8, so that's our output.

Let us say we have a training dataset $\mathbf{X} = (\{\mathbf{x}_1, ..., \mathbf{x}_N\}, \{t_1, ..., t_N)\})$. We will denote the set of training data indices belonging to node $\mathcal{T}$ as $I_\mathcal{T}$. In regression, our goal is to output some numerical value, so we can take

$$\hat{t}_\mathcal{T} = \sum_{i \in I_\mathcal{T}} t_i \tag{3.24}$$

as the prediction for this region. In classification, we can take the most frequent label of training data belonging to this leaf.

3.4.1 Creating a decision tree

We start by creating the root node, and assigning every training example to it. We want to split this node in a way that leaves us with a more homogeneous space partition than the one we currently have. For this we use something called a *criterion*. This is a function that tells us how homogeneous a collection of vectors is. In regression, we can simply use the sum of squares error between the prediction of a node, and the training examples belonging to it.

$$c_{SE} = \sum_{i \in I_\mathcal{T}} (t_i - \hat{t}_\mathcal{T})^2 \tag{3.25}$$

In classification, there are two popular choices - *Gini impurity*, and *Entropy*. Before looking at the formulas, note that the probability of class k in region $\mathcal{T}$ is $p_\mathcal{T}(k)$. Gini impurity is defined as follows

$$c_{Gini}(\mathcal{T}) = |I_\mathcal{T}| \sum_k p_\mathcal{T}(k)(1 - p_\mathcal{T}(k)) \tag{3.26}$$

It measures how often a randomly chosen element would be labeled incorrectly, if it was chosen according to $\mathbf{p}_\mathcal{T}$. The entropy criterion is defined as

$$c_{Entropy}(\mathcal{T}) = |I_\mathcal{T}| \cdot H(\mathbf{p}_\mathcal{T}) = -|I_\mathcal{T}| \sum_{k, p_\mathcal{T}(k) \neq 0} p_\mathcal{T}(k) \log p_\mathcal{T}(k) \tag{3.27}$$

Where $H(\mathbf{p}_\mathcal{T})$ is the entropy of the distribution $\mathbf{p}_\mathcal{T}$. When creating the tree, we usually have some constraints, which can be set as hyperparameters:

- maximum tree depth - don't split nodes with this depth
- maximum number of leaves - split until we reach this number of trees
- minimum examples to split - we can only split nodes which have more than this amount of training examples
- minimum criterion decrease - we only split the node if the sum of impurities (or other criteria) is less than the parent's impurity by at least this amount

If we don't have a limit on the number of leaf nodes, we split the nodes depth-first until we break some other constraint. If we do have a limit, we split the node which will result in the largest criterion decrease.

3.4.2 Model ensembling and Bagging

Decision trees on their own are quite weak, in the sense that their accuracy is not much better than random guessing. However, we can use them as base for an *ensemble* to get a more robust model. Ensembling is a technique in which we train multiple instances of a base model, usually each on slightly different training datasets, and use all of them during prediction. We can, for example, take the average result of all models in regression, or take the most frequent label in classification.

It can be proven that if we have M models, the average error of the ensemble is $\frac{1}{M}$ times the average error of the individual models. We denote the individual errors of the models as $\varepsilon_i(\mathbf{x})$, and the prediction of a model on a training example $(\mathbf{x}, t)$ as $y_i(\mathbf{x}) = t + \varepsilon_i(\mathbf{x})$, then the mean square error of the model is

$$\mathbb{E}[(y_i(\mathbf{x}) - t)^2] = \mathbb{E}[\varepsilon_i^2(\mathbf{x})] \tag{3.28}$$

Assuming the errors have zero mean and $\forall i, j : cov(\varepsilon_i, \varepsilon_j) = 0$, then

$$\mathbb{E}[\varepsilon_i(\mathbf{x})\varepsilon_j(\mathbf{x})] = 0 \tag{3.29}$$

and so

$$\mathbb{E}\left[\left(\frac{1}{M}\sum_i \varepsilon_i(\mathbf{x})\right)^2\right] = \mathbb{E}\left[\frac{1}{M^2}\sum_i\sum_j \varepsilon_i(\mathbf{x})\varepsilon_j(\mathbf{x}))\right] = \frac{1}{M}\mathbb{E}\left[\frac{1}{M}\sum_i \varepsilon_i^2(\mathbf{x})\right] \tag{3.30}$$

The uncorrelated condition is a strong one, and in practice it will not be so, but ensembling still gives better results than singular models. As was said before, we train the individual models on a bit different dataset precisely to lessen the correlation, and we can think of this as the models having a bit of a different perspective on the data. For this we used a technique called *Bootstrap Aggregation*, or bagging for short. The bootstrap part means that, before training, we sample the original dataset randomly using the empirical distribution, but with replacement. We sample as many indices as the original dataset, or we can use subsampling and only have a fraction. We can also choose random subsets of features, this is then called *Random Subspaces* [24]. Models which use decision trees in this way are called *Random Forests*.

3.4.3 Gradient Boosting Decision Trees

Random forests use a lot of decision trees trained independently to construct a more robust model. Instead, we could use the already trained trees to improve training of new ones. This approach is called *boosting*. In this case, we will be talking about *Gradient boosting*, specifically the method described in this paper [25].

Previously we talked about SGD, which used a learning rate set by us as a constant, to find the global/local minimum. This is called a *first-order method*. We could instead use *second-order method*, which computes the second derivative of the objective function, and uses it as a kind of adaptive learning rate. The resulting update in the SGD would look like this:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{f'(x)}{f''(x)}$$

This kind of update puts us a lot closer to the minimum in each step, however, computing the second derivative of the objective function is very inefficient for many applications. That's because if we have a vector of weights $\mathbf{w} \in R^D$, then the second order derivatives will consist of a matrix $D \times D$. However, with decision trees we can use an approximation. This results in a better splitting criterion, which is used during tree creation. Specifics can be found in [25]. An algorithm for finding the best split is also provided in that paper, which can be found in Algorithm 6.

Input: example set of the current node $I_{\mathcal{T}}$
Input: Feature dimension D
$score \leftarrow 0$;
$G \leftarrow \sum_{i \in I_{\mathcal{T}}} g_i$;
$H \leftarrow \sum_{i \in I_{\mathcal{T}}} h_i$;
for $k = 1$ **do**
  $G_L \leftarrow 0$;
  $H_L \leftarrow 0$;
  for j *in sorted* $I_{\mathcal{T}}$ *by* $\boldsymbol{x}_{jk}$ **do**
    $G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$;
    $G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$;
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$;
  end
end
Output: Split with max score

Algorithm 6: Greedy algorithm for Split Finding [25]

Other than feature and data subsampling we can also use shrinkage, which scales newly the predictions of newly added trees by a factor $\ni$, which gives the model more room to improve by reducing the influence of individual trees.

This works well for regression. For K-class classification we have to model the whole distribution, so for each class we have to train a separate "chain" of models, and use softmax on the combination of their results.

3.4.4 AdaBoost

Another thing we can do is to look at the samples which are most often misclassified during training. We could give them more weight in the next iteration, hopefully improving the accuracy of our model. The final model will be a linear combination of all the previous models, weighted by their error rates (probabilities of misclassification).This is the idea behind AdaBoost, described in [26]. This algorithm was designed for binary classification, and it works very well in that setting. It relies on the assumption, that the weaker models' error rate is $\leq \frac{1}{2}$. The pseudocode can be found in Algorithm 7.

$I(y_t(\mathbf{x}_i) \neq t_i)$ is an indicator $(= 0/1)$ whether the prediction of model M_t when given example $\mathbf{x}_i$ matches its target. In a K-class setting however, the random guess error rate is $\frac{K-1}{K}$, so it is much harder to achieve the needed error rate for weaker models. We are going to use a modification of this algorithm called

Input: Training dataset $\mathbf{X} = \{(\mathbf{x}_1, ..., \mathbf{x}_N), (t_1, ..., t_N)\}$
Input: Distribution $\mathbf{P}$ over the examples
Input: Weak model M
Input: Number of iterations T
$w_i^1 \leftarrow P(i)$, for $i = 1, ..., N$; `// Initialize the weight vector`
for *t = 1, 2, ..., T* **do**
  $\mathbf{p}^T \leftarrow \frac{\mathbf{w}^t}{\sum_{i=1}^N w_i^t}$ `// Normalize weights`
  Fit model M_t to training data with weight dist. $\mathbf{p}^t$;
  `// Calculate the error of this model`
  `// Prediction of this model is denoted as` y_t
  $\varepsilon_t \leftarrow \sum_{i=1}^N p_i^t I(y_t(\mathbf{x}_i) \neq t_i) / \sum_{i=1}^N p_i^t$;
  $\alpha_t \leftarrow \log \frac{\varepsilon_t}{1-\varepsilon_t}$;
  `// Set new weights`
  $w_i^{t+1} \leftarrow w_i^t \cdot \exp\left(\alpha_t \cdot I(y_t(\mathbf{x}_i \neq t_i))\right)$, for all $i = 1, ..N$;
end
Output: Model: $y(\mathbf{x}) = \arg\max_k \sum_{t=1}^T \alpha_t \cdot I(y_t(\mathbf{x}) = k)$
Algorithm 7: AdaBoost [26]

SAMME [27], which stands for *Stagewise Additive Modelling using a Multiclass Exponential loss function* - quite the mouthful.

Input: Training dataset $\mathbf{X} = \{(\mathbf{x}_1, ..., \mathbf{x}_N), (t_1, ..., t_N)\}$
Input: Distribution $\mathbf{P}$ over the examples
Input: Weak model M
Input: Number of iterations T
$w_i^1 \leftarrow P(i)$, for $i = 1, ..., N$; `// Initialize the weight vector`
for *t = 1, 2, ..., T* **do**
  $\mathbf{p}^T \leftarrow \frac{\mathbf{w}^t}{\sum_{i=1}^N w_i^t}$ `// Normalize weights`
  Fit model M_t to training data with weight dist. $\mathbf{p}^t$;
  `// Calculate the error of this model`
  `// Prediction of this model is denoted as` y_t
  $\varepsilon_t \leftarrow \sum_{i=1}^N p_i^t I(y_t(\mathbf{x}_i) \neq t_i) / \sum_{i=1}^N p_i^t$;
  $\alpha_t \leftarrow \log \frac{\varepsilon_t}{1-\varepsilon_t} + log(K-1)$;
  `// Set new weights`
  $w_i^{t+1} \leftarrow w_i^t \cdot \exp\left(\alpha_t \cdot I(y_t(\mathbf{x}_i \neq t_i))\right)$, for all $i = 1, ..N$;
end
Output: Model: $y(\mathbf{x}) = \arg\max_k \sum_{t=1}^T \alpha_t \cdot I(y_t(\mathbf{x}) = k)$
Algorithm 8: SAMME [27]

The only difference is in the computation of term α_t, namely the addition of $\log(K-1)$. However, this makes all the difference. This term makes it so that for $\alpha_t > 0$, we only need $(1 - \varepsilon_t) > \frac{1}{K}$, or in other words the accuracy of the weak models to be better than random guessing, rather than $\frac{1}{2}$. Pseudocode is in Algorithm 8.

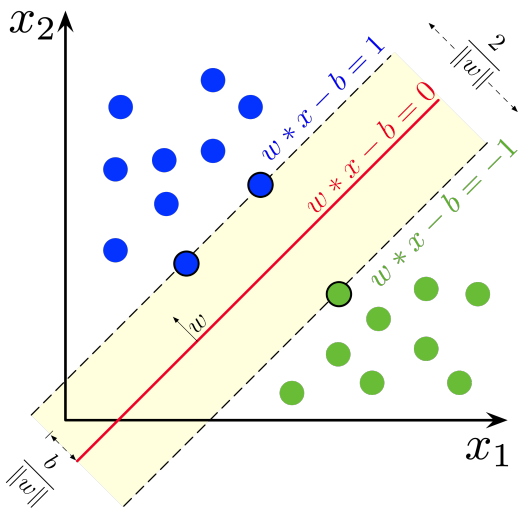


Figure 3.8: Hyperplane and margin visualization, from [28]. w is a vector of weights, $||w|| = 1$, x is a training example, b is bias, $*$ is vector multiplication. Data is classified into two classes based on the sign of the expression $w*x-b$. We can see that examples which lie on the margin have the value of that expression equal to 1 or -1, and every other example has a higher absolute values. In a soft margin SVM, some examples would be permitted to be inside the margin, i.e. $|w * x - b| \leq 1$.

3.5 Support Vector Machines

The theory included in this section is from Chapter 7 of [12], unless stated otherwise.

Support vector machines (SVMs) can be thought of as a kind of enhanced linear model. Their objective is to find a hyperplane which separates data into classes. A hyperplane is a subspace whose dimensionality is one lass than its ambient space. So for a 2D space it is a line, for 3D space it is a 2D surface and so on.

In contrast to, for example, a Perceptron, SVMs try to find a hyperplane with the maximum *margin*, which is the smallest distance between any training example and the hyperplane. This is done to achieve more robust results, since such a hyperplane tends to be a better estimate of the real boundary.

This version is called a *Hard-margin* SVM. However, it requires the original dataset to be linearly separable. In the Hard-margin version, the only way to alleviate this, is to project the input data into a higher-dimensional space, which might the hyperplane easier to find.

There is also a type of SVMs called *Soft-margin* SVM, which permits some training examples to be misclassified, if they are not too far away from the margin.

SVMs utilize functions called *kernels*, which allow them to efficiently compute polynomial combinations of features in vectors in our training data. These kernels are the aforementioned projections of input data into high-dimensional spaces.

For example, let our training examples consist of D dimensional vectors $\mathbf{x} = (x_1, ..., x_D)$. Let us say we have a transform function

$$\varphi(\mathbf{x}) : R^D \rightarrow R^{D'}$$

That maps the input vectors to a different dimensional space. For example, we could have a transform that outputs a vector together with its quadratic and linear features, i. e.

$$\varphi(\mathbf{x}) = (1, x_1, x_2, ..., x_D, x_1^2, x_1x_2, ..., x_2x_1, ..., x_Dx_{D-1})$$

The polynomial features might be beneficial for model performance, since they help the model make decisions based on combinations of features, as opposed to singular feature values.

In general, we can have features consisting of any degree of polynomial combinations of feature. A kernel corresponding to a transform is a function

$$K(\mathbf{x}, \mathbf{y}) = \varphi(\mathbf{x})\varphi(\mathbf{y}) \tag{3.31}$$

Using a kernel is beneficial, since if we want to compute polynomial features of degree d, we can use

$$K(\mathbf{x}, \mathbf{y}) = (\gamma \mathbf{x}^T \mathbf{y})^d \tag{3.32}$$

Where γ is a free parameter. This is called a *polynomial kernel of degree d* Similarly, if we want all the polynomial features of degree at most d, we can use

$$K(\mathbf{x}, \mathbf{y}) = (\gamma \mathbf{x}^T \mathbf{y} + 1)^d \tag{3.33}$$

which is called a *non-homogeneous polynomial kernel*. This is useful, because we can compute these features in $O(D)$ instead of $O(D^d)$ for a pair of vectors. We call this the *kernel trick*. Another popular kernel to use is called the *Radial Basis function kernel*, or RBF, which is defined as

$$K(\mathbf{x}, \mathbf{y}) = e^{-\gamma||x-y||^2} \tag{3.34}$$

where γ is once again a free parameter. This corresponds to a combination of polynomial kernels of all degrees $d \geq 0$.

3.5.1 Multiclass classification

Since SVMs are binary classifiers, if we want to use them for K-class classification problems, we have to get a bit creative. Typically, we use one of two schemes: one-versus-rest or one-versus-one.

The former constructs K binary classifiers, where each one tries to separate its class from all the other classes. For the output, we choose the with the highest probability from its classifiers' output. However here we need to modify the models to output probabilities instead of class labels.

The latter one constructs binary classifiers for each pair of classes, and during prediction we choose the class with most votes.

In both cases we sacrifice some accuracy however, since there always exists a part of input space which will be ambiguous for all classifiers.

3.6 Preprocessing techniques

Input preprocessing is a key part of training ML models that greatly enhances performance. Here, we will describe some preprocessing techniques used during model training.

3.6.1 TF-IDF

Term frequency - Inverse Document Frequency (TF-IDF)[29] is an often used statistic to create a numerical representation of a document using token frequencies. Tokens are usually words n-grams, or word n-grams. n-grams are simply windows of text of length n. For example, let us say that $n = 6$, and we want character-level n-grams. Then, from the sentence "Hello world!" we would extract the following 6-grams:

```
"Hello ", "ello  w", "llo wo", "lo wor", "o worl",
" world", "world!"
```

As the name suggests, TF-IDF consists of two parts:

Term frequency

This is simply the relative frequency of a token t w.r.t. a document d.

$$\mathrm{TF}(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \tag{3.35}$$

Here $f_{t,d}$ is the frequency of token t in document d, i.e. the number of occurrences of this token in the document. Since it is a relative frequency, we simply divide $f_{t,d}$ by the sum of all token frequencies in the document.

Inverse document frequency

IDF measures how often a token appears in the document corpus D. It is defined as

$$\mathrm{IDF}(t, D) = \log \frac{N}{|\{d \in D; t \in d\}|} \tag{3.36}$$

We use a logarithm because this statistic is related to a concept called self information, which we will not be covering in this thesis.

To calculate IDF, we have to know the total frequency N of the token in the whole document corpus D, and also the number of documents that contain token t. Thus, before the calculation, we need to construct a vocabulary of tokens over the whole corpus. This vocabulary will contain total number of occurrences of every token t. In addition, we also need a data structure which will hold the frequency of each token t in a given document d.

Finally, we define TF-IDF as

$$\text{TF-IDF}(t, d, D) = \mathrm{TF}(t, d) \cdot \mathrm{IDF}(t, D) \tag{3.37}$$

So it is a statistic that, given a token t, document d and document corpus D, produces one real number. We can derive TF-IDF using information theory as well, since it is a form of mutual information between the token t and document d, but we will omit it here. TF-IDF works well, because it represents the amount of information learned by the presence of t in document d.

As stated previously, we can use TF-IDF to obtain a numerical representation of a document. This is quite handy, since most models take vectors of real numbers on the input. First, we order the tokens in the vocabulary. Then, for

```
corpus = ["ML is the best", "To the moon"]
ordering = { "best" : 0, "is" : 1, "ml" : 2,
          "moon" : 3, "the" : 4, "to" : 5 }
tfidf vectors = [
[0.53404633, 0.53404633, 0.53404633, 0.    , 0.37997836, 0.     ],
[0.      , 0.      , 0.      , 0.6316672, 0.44943642, 0.6316672]]
```

Figure 3.9: TF-IDF input transformation example. The corpus of two documents, in this case sentences "ML is the best" and "To the moon", is transformed using TF-IDF into numerical vectors. Note that the length of the output vector is the same as the size of the vocabulary.

each token t_i, we calculate its TF-IDF statistic relative to document d. The numerical representation of document d will be a vector of the same length as the vocabulary that contains the value TF-IDF(t_i, d) at the i-th position. Lastly, we normalize each vector v using the euclidean norm

$$\frac{v}{||v||_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + ... + v_n^2}}$$

3.6.2 Principal Component Analysis

Principal Component Analysis, or PCA, is a technique used to reduce the dimensionality of data. There are many ways to derive PCA, and we will use the covariance method, adapted from [30]. First, for given two real valued random variables X, Y, the covariance is defined as

$$cov(X, Y) = \mathbb{E}\left[(X - \mathbb{E}[X])(Y - \mathbb{E}(Y))\right] \tag{3.38}$$

If we have measurement vectors $\mathbf{x} = (x_1, ..., x_n)$ for X and $\mathbf{y} = (y_1, ..., y_n)$ for Y, the covariance is calculated as

$$cov(X, Y) = \frac{1}{n}\sum_i x_i y_i = \frac{1}{n}\mathbf{x}\mathbf{y}^T \tag{3.39}$$

If $cov(X, Y) = 0$, then the two variables are uncorrelated (but not necessarily independent).

Suppose we have a feature matrix M with n rows and m columns. We can look at the i-th column as a collection of measurements of a random variable, which represents the i-th feature. Then, we can construct a covariance matrix C_M of size $m \times m$ as follows:

$$C_M = \frac{1}{n}MM^T \tag{3.40}$$

A diagonal entry $C_{i,i}$ represents the variance of the i-th feature. All other entries $C_{i,j}$ represent the covariance between i-th and j-th feature. We assume that high variance of a variable means that there is potentially a lot of information to be gained from it. On the other hand, a high covariance between two variables

```
corpus = ["ML is the best", "To the moon",
          "I like the stock"]
result = [[ 7.67536453e-01,  1.39571107e-16],
          [-3.83768227e-01, -6.52490885e-01],
          [-3.83768227e-01,  6.52490885e-01]]
```

Figure 3.10: PCA transformation example. Vectors in the figure were obtained by transforming the TF-IDF features obtained from the corpus using PCA with two principal components.

means that there is some redundancy. Preferably, we would want to transform the matrix such that the variance is maximized, and redundancy eliminated. We can achieve this with diagonalization of the covariance matrix C_M. By using such transformation, we would obtain wholly uncorrelated features.

To calculate this transformation, we can compute the eigenvectors of C_M. This is thanks to the fact that C_M is symmetric, and it can be diagonalized by an orthogonal matrix P of its eigenvectors as columns. We can write it in an equation as follows

$$C_M = P^T D P \tag{3.41}$$

where D is a diagonal matrix containing eigenvalues of C_M. We can sort the eigenvectors according to the sizes of their corresponding eigenvectors, in descending order. Then we reorder all matrices to respect this ordering, and we have obtained our transformation. The eigenvectors are now called principal components, and represent the directions along which the variance is maximized in the original feature space.

Finally, we can select only the first n principal components, and use only them in the transformation, obtaining a projection of the data into a lower-dimensional space. Naturally, we loose some information by doing, but since we ordered them according to decreasing variance, we always retain as much information as possible. This, of course, assumes that a large variance, and therefore large eigenvalue, means a lot of useful information contained in a given feature.

In practice, instead of eigenvector decomposition, we can use other methods such as Singular value decomposition (SVD) and even approximate methods such as Randomized SVD. They all result in some representation of the data that reduces dimensionality while maximizing variance. When we used PCA in the experiments, we used a full SVD.

4. Analyzed Social Network Data

This chapter contains description of datasets used for the practical part of the thesis.

4.0.1 Dataset Descriptions

We have scraped data from Reddit and Twitter using freely available libraries *snscrape*[31] and *PSAW*[32]. This data contains posts and tweets which included at least one instance of words "GME" or "GameStop" (both can be case-insensitive).

Reddit

In case of Reddit, we only gathered data from subreddit r/wallstreetbets, which was the subreddit where a lot of the community discussion took place [33].

This data spans approximately one year. It begins at January 1st, 2021 and ends at February 2nd, 2022. The beginning was chosen so that it included a few weeks before the major spike of GameStop's stock price, on January 28th, 2021.

There is a lot of information available about each post. For Reddit, we get

- id - Unique post ID
- selftext - Text content of the post
- title - Title of the post
- created_utc - Timestamp of the post creation time
- score - Number of upvotes minus number of downvotes
- upvote_ratio - Ratio of upvotes and downvotes
- num_comments - Number of comments of the post

	score	selftext	title	created_utc	upvote_ratio	num_comments	author
sn3e06	1	\n\n\n\nRite Aid stock (RAD) has all the ingre...	Are t Yahoo Message boards hacked and controll...	1644273849.0	0.51	37	{'name': 'JazzPlayer77', 'id': 'ao4y8mxa'}
sn2a5c	8431	We all know FAANG (Facebook, Amazon, Apple, Ne...	You've heard of FAANG, but have you heard of L...	1644271201.0	0.93	508	{'name': 'ISayBullish', 'id': 'at54fjaq'}
smvecp	1		Is gme still happening?	1644254195.0	1.0	1	{'name': 'Dreadluxx', 'id': '13vasyg4'}
smf0cy	0	Hear me out guys,\n\nFuckbook sucks ass we all...	Were in a bear market	1644201508.0	0.27	34	{'name': 'GoodLife4life', 'id': 'jf1z1'}
slim48	1		AMC/GME question regarding hedge funds	1644100151.0	1.0	2	{'name': 'lletthedogsoutmeme', 'id': 'gmzvkil'}

Figure 4.1: Reddit dataset in a DataFrame [34]

Title: Broke Bois
Text: For those of us who lack the nugs to pony up anymore on the big rocket what can we do to stick it to the hedge funds even more?

I can't drop anymore bread into GME at this point. Out of my price range and I think so broke bois can agree. I've been dipping into SNDL trying to see if this pint sized size stock can at least make it into orbit. I've got enough to make small gains there. Not enough to buy a telsa but maybe some bags of frozen tenders.

What do y'all think?

(Anonymous Reddit user)

Figure 4.2: Reddit post example. Many of the posts are much, much longer than this

Title: GME First FALSE DIP. HOLD YOU SAVAGES.
Text: HOLD THE LINE! They're creating a false dip!

(Anonymous Reddit user)

Figure 4.3: Another Reddit post example. Some of them are shorter, like this one.

On Reddit, user can "upvote" or "downvote" a post, based on whether they agree with it or not. The total difference between these is called the post "score". If there are more upvotes than downvotes, the score is positive, and in the reverse case it is negative. Naturally this can be used when we want to estimate the sentiment of the community in relation to the content of the post. Here we can see a few examples of posts in r/wallstreetbets:

There is a caveat however, and that is that a lot of the posts might have been deleted by the user or by a moderator. In this case, the content of the post will usually just be a string "[deleted]". Because of this, they are not useful for our analysis. The data that we have scraped will provide only an estimation of the community's sentiment. We are also only looking at posts that had text in them. Many times, people will post screenshots or memes, and our models do not focus on analyzing visual data.

Twitter

A lot of discussion about GameStop has also been happening on Twitter, so we also scraped any tweets containing "GME" or "GameStop" during the same timeframe. For this, we have used the service *snscrape* [31], which scrapes and archives posts from many social networks, including Twitter.

For every Tweet we have the following fields

- Date - Date of Tweet posting
- UserID - ID of the Tweet author

	Date	UserID	Content	ID	LikeCount	ReplyCount	RetweetCount	QuotedTweet	RetweetedTweet
0	2022-02-07 21:39:59	1377438365327429633	Needing some follows...set notifications for upd...	1490802544834224128	23	3	5	1.490802e+18	NaN
1	2022-02-07 20:13:47	1457625929841315841	#gamestop $gme GameStop Ryan Cohen's vision is...	1490780853563273221	3	0	2	NaN	NaN
2	2022-02-07 20:04:05	173983233	@knotty_llama #gme @GameStop #powertotheplayers	1490778411786289154	1	0	0	NaN	NaN
3	2022-02-07 19:52:10	1431049311006216193	Immutable X's President seems like the real de...	1490775412988858370	2	0	0	1.490724e+18	NaN
4	2022-02-07 19:36:29	244181007	If you want change in the financial markets, d...	1490771464789712899	1	0	0	NaN	NaN
...	...	...	...	...	...	...	...	...	...

Figure 4.4: Twitter dataset in a DataFrame [34]. The RetweetedTweet column also includes actual Tweet ID's, the values are just written in a different way by the visualization. NaN values are there, if there isn't any QuotedTweet or RetweetedTweet in this Tweet.

> `@andriusdaulys @CNNBusiness where did you source your info re silver being the hottest thing now? $gme #gme #gamestop - you're manipulating the market if you have nothing to back this up besides the fact that citadels portfolio is heavy on SLV. Zero integrity`
>
> (Anonymous Twitter user)

Figure 4.5: Tweet example

- Content - Textual representation of the tweet content
- ID - unique Tweet ID
- LikeCount - Number of likes
- ReplyCount - Number of replies
- RetweetCount - Number of retweets
- QuotedTweet - ID of the Tweet quoted by this one (if it exists)
- RetweetedTweet - ID of the retweeted Tweet by this one (if it exists)

One benefit of Twitter data is, that the text is a lot shorter than Reddit. It can be much more easily annotated. Also, we can use them to create a network of Twitter users, as stated in the second objective. More on how to do this in a later section.

Enron

Enron was a major corporation operating during the late 20th and the beginning of 21st century. It went bankrupt because of a major corruption scandal involving most of the upper management. During the court proceedings, a large amount

	$\bar{x}$	σ	Q_{25}	Q_{50}	Q_{75}	x_{max}
Score	746.4	4344.0	8.0	24.0	127.0	143619
Volume	29.3	231.8	1.0	3.0	7.0	3603
Length	1089.8	2231.8	86.0	338.0	1153.8	39925

Table 4.1: Reddit statistic table. $\bar{x}$ = mean value, σ = standard deviation, Q_x = x-th quantile, x_max = maximum value. Volume is post volume daily, Length is text length in characters.

of corporate emails have been publicized. Today, this email dataset is freely available online [35].

Naturally, we can use emails to easily create a network. Since this network is a lot larger than the Twitter one, we can test our structural analysis on this one, and later use it for Twitter as well.

The emails themselves have some basic SMTP based headers, the body of the email, and also some metadata written by people who compiled them. This metadata mostly contains the real names of people, to which the email addresses belong. However, this metadata is non-homogeneous, specifically the formatting isn't. So for our purposes, we used email addresses in the SMTP headers for user identification. Some users have had multiple addresses, however this is a compromise we have to make, because of the size of the dataset.

4.1 Dataset analysis and visualization

In this section we will see some interesting visualizations made from the data in the previously mentioned datasets.

4.1.1 Reddit

First, we can look at some general information in table 4.1. At a first glance, the deviations in every metric are quite large. For post volume and score, we can see that because of difference between Q_{50} and Q_{75}, most posts were very unpopular, and there were a few extremely popular ones. Similarly, most days had almost no posts, and a few had hundreds.

The mean post length is also quite long. This is because a lot of people make lengthy posts about their trading strategies. These posts usually involve lengthy company analyses and a heavy argumentation for their chosen stock. Sometimes they are about GameStop, oftentimes not. But those that aren't usually predict that their stock "will be the next GameStop", and so they're included in the dataset.

Then, we can plot information about volume and average score of a post during the time period (Figure 4.6).

We can see that the greatest volume of posts is from mid-January until late March 2021. This makes sense, since the GME short-squeeze happened at the end of January. Community hype picked up after this. Stock price started to fall however, but other major spikes happened in the middle of march and beginning of June, and we can see the corresponding increase in volume during those periods.

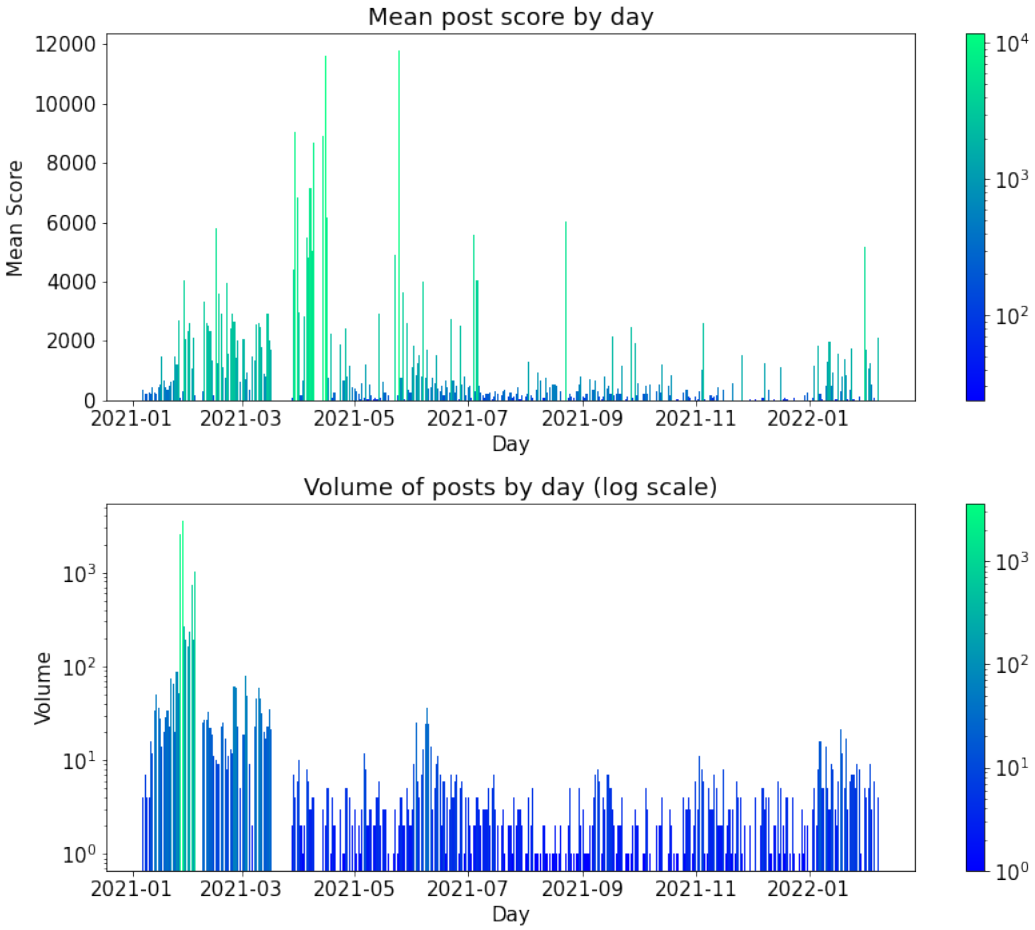


Figure 4.6: Reddit daily data. Data is colored so that everything below the 25th quantile has the darkest color. The volume plot has a logarithmically scaled y-axis, since the differences between post volumes are very large.

Interestingly, even when there were much fewer posts in the later months, the mean score of them was quite high. We can more easily see this in a monthly chart (Figure 4.7)

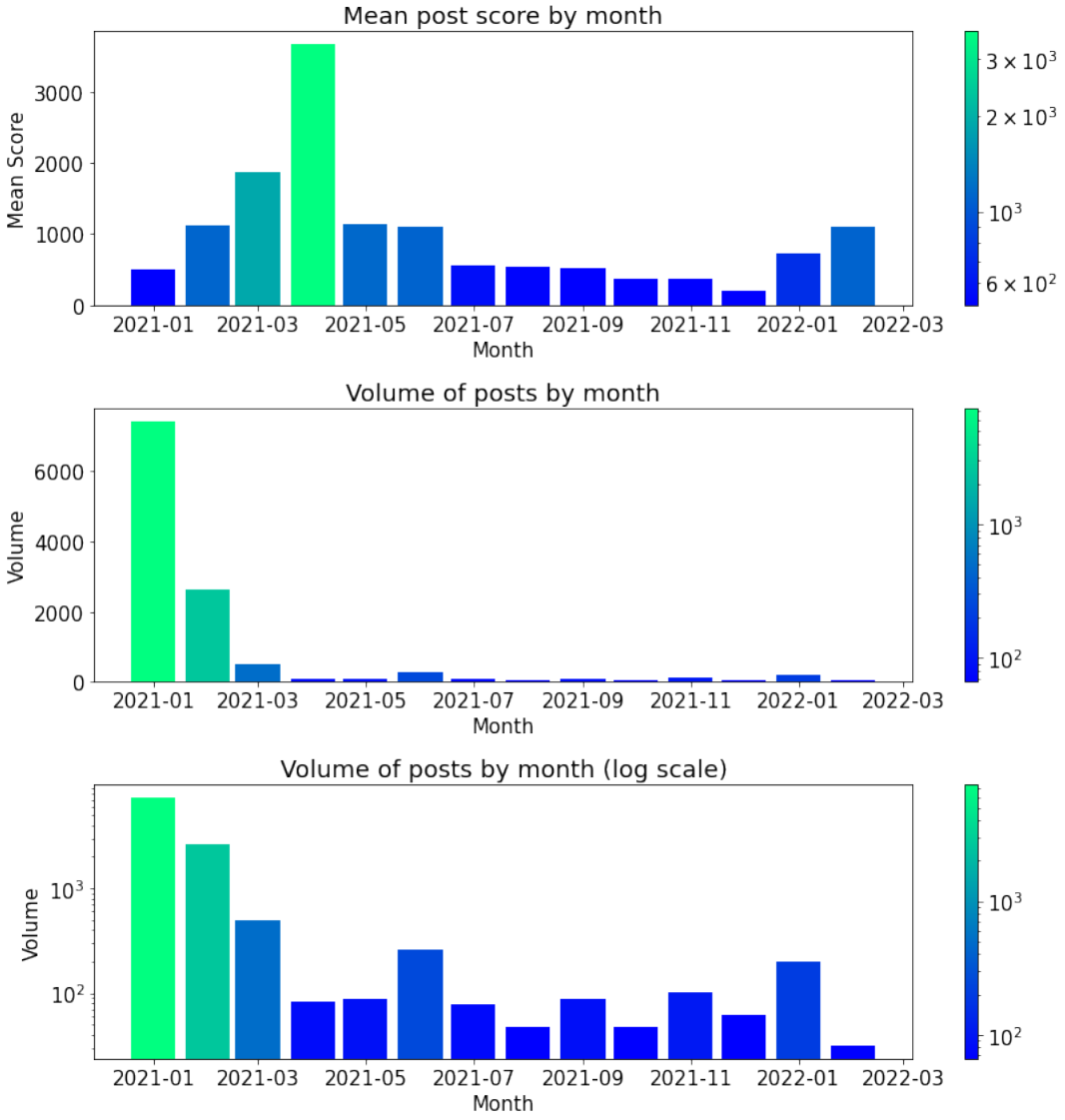


Figure 4.7: Reddit monthly data. Coloring has the same rules as previously.

Notice the large drop-off in later months. This might be explained by the fact that the community has moved on, and partially because other GameStop-focused subreddits have been created, and the most fanatical users moved there.

4.1.2 Twitter

Data Analysis

We can look at Twitter data similarly. The difference in respect to Reddit is, that users cannot downvote Tweets, only like them. They can also retweet them, which helps to spread their popularity. This 4.2 is the table containing general information about Twitter data. We can see that while the Like and Retweet

	$\bar{x}$	σ	Q_{25}	Q_{50}	Q_{75}	x_{max}
Retweet count	1.4	15.2	0.0	0.0	1.0	1302
Like count	10.9	109.7	0.0	1.0	4.0	11424
Tweet volume	68.1	180.2	23.0	37.0	57	2713
Tweet length	149.2	86.1	81.0	129.0	208.25	965

Table 4.2: Twitter general information. Tweet length is the number of characters in a Tweet.

counts are a lot lower than Reddit post score, the average volume is more than two times higher. Also, since Twitter has a character limit on posts, the average length of a Tweet is smaller than of a Reddit post, especially in Q_{75}. This makes Twitter a lot easier to annotate. Another interesting thing is that a great majority of tweets have little to likes and retweets. This, combined with the high standard deviation suggests that there exists a minority of very popular tweets.

Figures with daily and monthly Tweet data can be found here 4.8 and here 4.9. Unsurprisingly, the greatest volume of Tweets was around the same time the spike in Reddit posts. What is surprising, is that as opposed to Reddit, the Retweet and Like counts seem to be steadily going up as time passes. My hypothesis is that over time, a few Twitter accounts related to GameStop have gained a cult-like following from dedicated communities on Reddit and elsewhere, and they began to engage with most of their new Tweets.

Since the volume of Tweets in the later months is much smaller than at the beginning, these tweets from popular accounts account for a bigger part of the total volume, and thus prop up the mean Like and Retweet counts.

Network Analysis

On Twitter, since users can follow or retweet each other the data can be used to construct a network of these users. Each node will be a user, and directed edges between them will represent their relationships. So if A followed or retweeted B, there will be an edge (A, B) with weight 1, and if A retweeted B multiple times, we add 1 to the edge weight for each retweet.

A sketch of the procedure can be found here 9. After computing this graph, we find that the Twitter network consists of 27285 nodes and 40745 edges. As discussed in 2.4, for every node, we are going to compute its PageRank, Hub and Authority scores, and its Betweenness Centrality. Additionally, we are also going to attempt to calculate a degree exponent γ for this network.

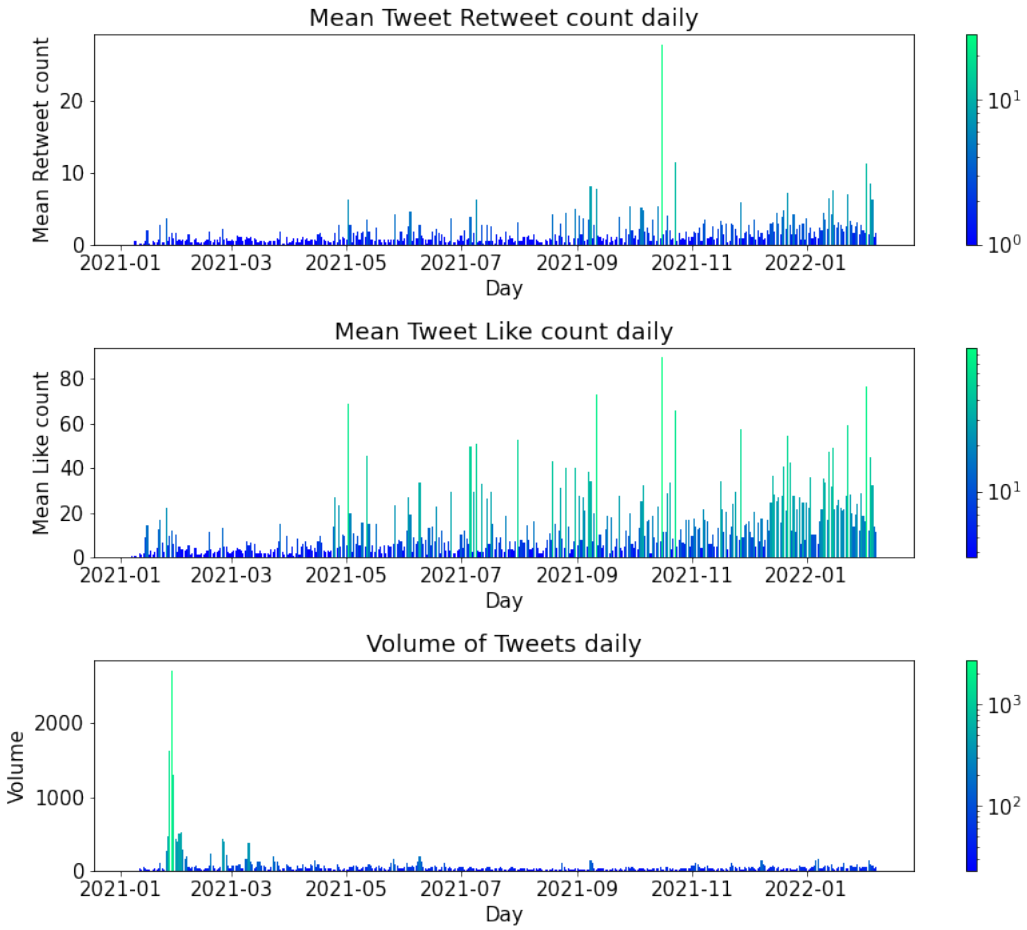


Figure 4.8: Twitter daily mean data.

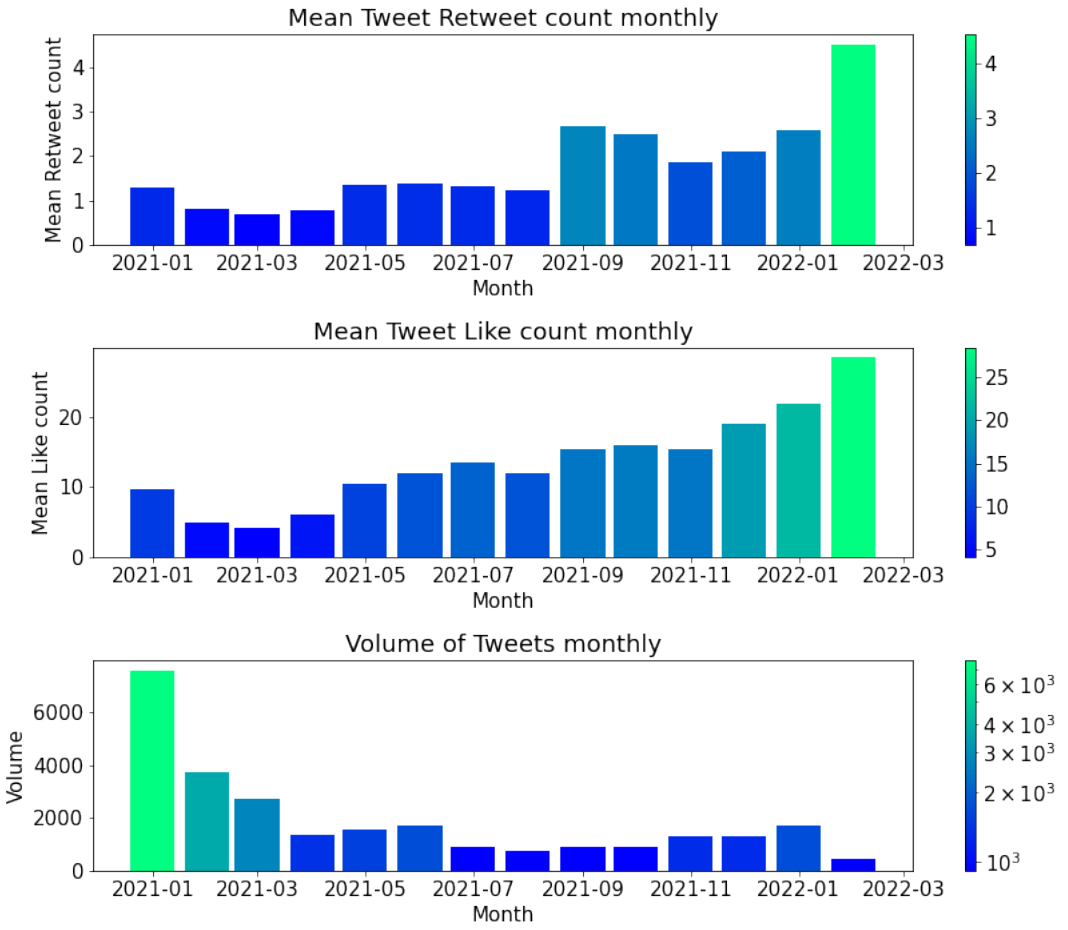


Figure 4.9: Twitter monthly mean data.

Input: Set of Tweets T
$G \leftarrow$ empty directed graph;
foreach *Tweet* $t \in T$ **do**
  $A \leftarrow$ author of t;
  if $A \notin V(G)$ **then**
    foreach *follower* F *of* A **do**
      insert edge (F, A) with weight $w_{FA} = 1$ into G;
    end
  end
  foreach *retweeter* R *of* t **do**
    if *edge* $(R, A) \in E(G)$ **then**
      $w_{RA} = w_{RA} + 1$;
    else
      insert edge (R, A) with weight $w_{ra} = 1$ into G;
    end
  end
end
Output: Network G

Algorithm 9: Twitter network creation

4.1.3 Enron

Lastly, we have the Enron dataset. We will not be delving deep into this dataset, even though it is also interesting, since the main focus of this thesis is the data from Reddit and Twitter. However, we will also estimate the various metrics, and calculate all the metrics and its degree exponent. This because we want to compare the results with the Twitter network. The graph created from this dataset is also much larger than the Twitter one, so we could potentially get more accurate results as well.

The Enron graph contains 87482 nodes and 323032 edges. This is after pruning 509 nodes, which did not contain email addresses in their name, and removing resulting nodes with degree of 0, of which there were 175. This was done to simplify data parsing, and does not have a large impact on the network, because we only removed small fraction ($< 1\%$) of edges.

As mentioned before, the dataset contains emails of Enron employees together with their SMTP headers. In some emails there are also notes of people who collected and processed these emails, marking real names of senders and recipients in some email addresses. However, formatting of these notes is not at all standardized, in some cases notes are missing, and in other cases the notes only have email addresses instead of names. This is why a decision was made to only take into account actual email addresses. It is known that some people had multiple addresses, which will skew things, but they should still be a good approximation.

4.2 Hypotheses

Now, we will outline some hypotheses that we would want to investigate via network and sentiment analysis.

4.2.1 GameStop

1. **Community sentiment should be generally positive**. As stated in the Introduction, GameStop's stock value was largely due to community hype. The r/wallstreetbets subreddit is also generally biased toward optimistic claims about discussed companies. This is done to sway potential investors and drive stock price upwards. Of course, there may also be bearish (expecting negative movement) posts toward companies, but from anecdotal experience this is not as frequent. Thus, we expect the community sentiment to be mostly positive.

2. **Ryan Cohen, Keith Gill and GameStop's Twitter account should be among the most important nodes in the network**. Ryan Cohen became GameStop's new chairman a few weeks between the spike in its stock value, and has been viewed as having a positive impact on the company. He is also active on Twitter, and often talked about in communities related to GameStop. His account should be one of the most important according to most metrics, except perhaps Hub score. This exception is because even when he is considered active, public figures such as himself do not interact nearly as much with other users ordinary users do.

   Keith Gill (Reddit username DeepFuckingValue, Twitter username Roaring Kitty) is also a famous GameStop-related figure, being one of the first people to make a strong positive case for the stocks value. He also made regular updates of his positions value on r/wallstreetbets, which ballooned from $53,000 to around $50,000,000 during the price peak. This, together with his positive personality, has made him beloved by the r/wallstreetbets community. His quote "I like the stock", when asked to testify in front of the U.S. House Committee on Financial Services following the short-squeeze, is still used today on r/wallstreetbets. His account should also be one of the most important ones, even if it became inactive in mid-2021.

   Finally, anecdotal experience from annotating data suggests that the GameStop account will also be among the top accounts. This is because many tweets, when talking about GameStop, also included a link to this account. Since we only have posts which include the word "GameStop" or "GME" in their textual content, this occurrence is often a mention. And since we count mention as edge-creating relations, GameStop will be a highly connected node in the network, and thus have a high score in most metrics.

4.2.2 Enron

1. **Enron's executives such as Jeff Skilling and Kenneth Lay should be ranked high in the importance scores.** Kenneth Lay was the founder and CEO of Enron, and was found guilty of securities fraud. Jeff Skilling was also the CEO, and was also convicted and sentenced to 24 years of prison. Both of these men were active in the company, and complicit in the fraud that was happening, and so they should be among the most important nodes in the network.

2. **Hierarchical corporate structure will lead to high betweenness scores for the middle management**. Middle management is, among other things, responsible for communication between different levels of the corporation. This means that they should often be on direct paths between distant parts of the network and people under their management. Betweenness centrality is based on the fraction of the shortest paths between other nodes passing through this node, and so middle management should have a naturally high betweenness.

5. Supporting experiments

In this chapter we will talk about the setup of the experiments, insights during training and the results we have obtained. As outlined in the Introduction, our three main goals are

1. Create a model for the sentiment of posts from Reddit and Twitter, and use it for analysis of the community sentiment over time.
2. Build a network graph from the Tweets and Enron emails and identify important nodes using methods from Section 2.4.
3. Investigate the plausibility of a power-law degree distribution for these networks using a goodness-of-fit test.

5.1 Experiment Setup

5.1.1 Sentiment model setup

We will be training various machine learning models described in Section 3. Models which are built upon neural networks(MLPs and RNNs) will be implemented using an open-source Python library *tensorflow* [36]. All the other models will be implemented using *scikit-learn* [37].

The models will be trained on a manually annotated dataset containing 1657 Reddit posts and Tweets. Each post is labeled either positive, negative or neutral, based on sentiment of the textual content of the post w.r.t GameStop. Internally, the labels are integers where positive = 1, negative = 0 and neutral = 2. Pie charts information about the dataset can be found in Figure 5.1.

Data is roughly equally split between Reddit posts and Tweets, with Tweets having a slight majority. This is because Tweets are generally easier to annotate. The data is heavily skewed towards the positive and neutral labels. This makes training accurate models more difficult, and if we don't take special measures, the models could learn to simply ignore the negative label.

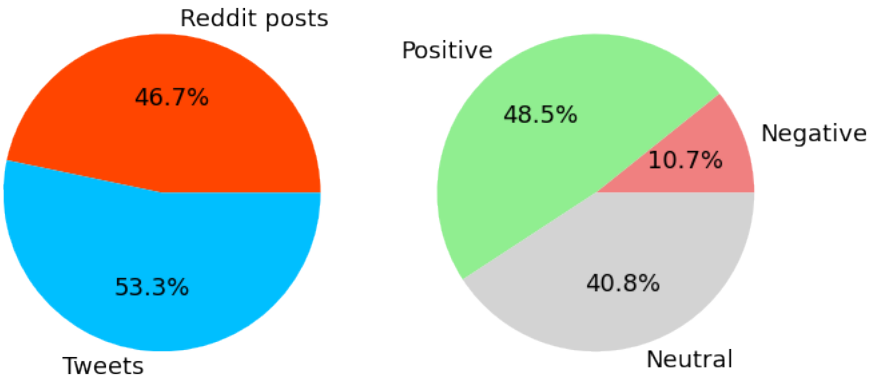


Figure 5.1: Information about the annotated dataset makeup.

5.1.2 Network parameter calculation setup

Power law parameter fitting and estimation will be done using an R language library called *poweRlaw*[38]. This library uses techniques described in [6] to estimate the scaling parameter and also the k_min. It also includes hypothesis testing using bootstrapping as described in the aforementioned paper.

5.2 Sentiment model results

In this section, we will discuss the performance of the models described in Section 3. During training, we have tried to optimize models to have as high macro-averaged F1-score as possible, and will be ranking them accordingly. We have also kept track of accuracy and other variations of the F-score. All models have been trained with random seed set to 42 if applicable, and 5-fold cross-validation.

Except for RNNs and some other specific cases, the preprocessing of the data was as follows:

1. **Clean the text** - Text is lowercased, punctuation and newline characters are ignored. Optionally, URLs can be replaced with "[link]".

2. **Convert the text of the post into TF-IDF features** - n-grams are extracted from the text, usually either using one n, or extracting all en-grams in range $[1, n]$. These can be word-level n-grams, or character-level n-grams. Using this vocabulary of n-grams, create the TF-IDF features. Set a minimum or maximum IDF value, so that very rare n-grams are excluded from the vocabulary. This reduces the dimensionality of the feature space and also acts as a denoising method.

3. **Use Principal Component Analysis (PCA) on the TF-IDF vectors, obtaining the final features** - The number of principal components is one of the hyperparameters as well.

Text cleaning is done to standardize the contents of the posts as much as possible. TF-IDF looks at a document as a bag of words, and we want to make sure that the data is not too noisy by having many variations of the same word. TF-IDF would, by itself, consider words "the" and "The" as completely different words. Capitalization of the first letter holds some information, for example that this might be the beginning of a sentence. However, it tends to muddle the text too much from a bag of words perspective, and tends to give worse results. Same principle applies to punctuation and sometimes braces as well. We have decided to remove the braces, but they could potentially improve the model too. If character-level n-grams are used, punctuation and braces becomes less of a problem, since the number of n-grams containing these characters is usually dwarfed by the version without punctuation. URLs have been transformed into "[link]" for the same reason, to not clutter the vocabulary with unique links. The specifics of the URL are usually unique to the post, but the knowledge that there was a URL in the document may be relevant, so that is why we have transformed them into "[link]".

TF-IDF vectors are as long as the size of the vocabulary used to make them, and so the dimensionality is extreme. Raw TF-IDF vectors created from the

annotated dataset using 1 to 3-grams have length of around 84000. By using PCA, we can drastically reduce the dimensionality and still keep a good level of accuracy.

Recurrent neural networks take a sequence of vectors as their input, so it does not make sense to use TF-IDF or PCA. We will instead create a vocabulary, transform the text into a vector of indices to this vocabulary, and pass it to the network. Thanks to the use of an embedding layer, we do not have to transform the indices into a one-hot representation, because it will internally represent them as such. Output of the embedding layer is a sequence of k-dimensional vectors of the same length as the input sequence (k is a hyperparameter). This gives each word its own fixed size representation, which may include some information as well. Weights for the embedding layer are trainable parameters, so the model will learn to create embeddings that lead to the smallest loss on the output.

5.2.1 Naive Bayes

Starting off with the simplest models, they have been surprisingly effective. We started by choosing the distribution we assume the features have come from. We will consider either the Multinomial or Gaussian distribution. For the Multinomial Naive Bayes, we can choose the strength of the Laplace smoothing (denoted α). In the Gaussian version, we can choose the portion of the largest variance of features that is added to all variances.

To demonstrate the effect of minimum DF, n-grams and text cleanup, we have made experiments without these techniques and with In Table 5.1, we can see that setting the minimum document frequency even as low as 2 provides a great improvement in performance. Lowercasing and cleaning the text also gives us nice bumps in performance, as we would expect given the reasons we have talked about before. Using both n-grams usually improves performance too, since the presence of a specific pair of words gives the model more information than it would have otherwise. However, it doesn't always have to help as we can see in the last two experiments. If overdone, it can also hamper the model by overloading it with features.

We can try different types of Naive Bayes (NB) estimators and see how they perform. Available options are either Multinomial NB, Gaussian NB or Complement NB [39]. We have chosen these three because of the nature of our data. Since the TF-IDF vectors are non-negative, we can use the Multinomial version. It interprets the data as word counts, not TF-IDF, but in practice it works well too. Gaussian NB assumes that a given feature is normally distributed, and hence it can handle negative numbers as well. This means we can also use PCA. Lastly, Complement NB is a variant of the Multinomial NB, which "uses statistics from the complement of each class to compute the model's weights". This means that during training, the weights for class i are calculated from all the documents *not* whose target is *not* i. Then, we choose the class whose sum of these complement feature weights multiplied by the feature values is the smallest. The results of fitting the various NB types are in Table 5.2.

Complement NB has proven to be way above others in F_1-Score, even though it lost some accuracy over Multinomial. We had to do some experimentation to find the right configuration of hyperparameters, but generally (1, 3) n-grams were

Description	Acc % (σ %)	F_1 (σ)	$F_1^{(w)}$ (σ)	$F_{0.5}$ (σ)
raw text	52.1 (1.0)	0.294 (0.012)	0.41 (0.014)	0.312 (0.016)
min_df = 2	54.8 (2.2)	0.339 (0.023)	0.467 (0.028)	0.359 (0.026)
min_df = 5	55.5 (2.2)	0.353 (0.018)	0.483 (0.023)	0.366 (0.022)
min_df = 10	57.0 (1.9)	0.378 (0.011)	0.514 (0.014)	0.387 (0.013)
lowercase	59.7 (2.0)	0.403 (0.016)	0.546 (0.018)	0.408 (0.016)
cleaned	**60.5** (2.7)	**0.412** (0.022)	**0.557** (0.028)	**0.416** (0.023)
cleaned*	60.3 (1.9)	0.411 (0.014)	0.556 (0.018)	0.414 (0.016)
cleaned**	59.6 (0.023)	0.406 (0.016)	0.549 (0.021)	0.408 (0.019)

Table 5.1: Preprocessing demonstration table. $F_1^{(w)}$ is the weighted F_1-Score. Results are for Multinomial Naive Bayes. All models have been trained with $\alpha = 1$ and word-level n-grams, starting with 1-grams. No PCA was applied. Raw text means no text preprocessing. "min_df" means pruning words whose document frequency (DF) is $\leq$ the given amount. "lowercase" is lowercasing the text before training, and min_df = 10. "cleaned" is ignoring braces punctuation and newline characters, lowercased text and min_df = 10. "cleaned*" is the same as "cleaned", but using 1- and 2-grams. "cleaned**" is as before, but also including 3-grams.

Type	Acc% (σ%)	F_1 (σ)	$F_1^{(w)}$ (σ)	$F_{0.5}$ (σ)
Multinomial	61.1 (2.5)	0.463 (0.029)	0.583 (0.024)	0.489 (0.047)
Complement	**61.7** (1.7)	**0.53** (0.015)	**0.612** (0.017)	**0.535** (0.15)
Gaussian	51.7 (3.2)	0.48 (0.033)	0.514 (0.03)	0.482 (0.033)
Gauss PCA	38.2 (1.3)	0.337 (0.019)	0.337 (0.021)	0.345 (0.031)

Table 5.2: Naive Bayes model results. Best configurations shows. Multinomial NB had $\alpha = 0.1$, cleaned text, min_df = 10, ngrams = (1,2). Complement NB had the same configuration. Gaussian NB had variation smoothing = $4.641 \cdot 10^{-}3$, cleaned text, min_df = 5, ngrams = (1,2). Gaussian PCA had variation smoothing = $10^{-}4$, min_df = 5, ngrams = (1,2), and number of components for PCA = 200.

Type	Acc% (σ%)	F_1 (σ)	$F_1^{(w)}$ (σ)	$F_{0.5}$ (σ)
Non-PCA (1,4)	61.4 (1.4)	0.467 (0.032)	0.587 (0.16)	0.489 (0.05)
Non-PCA (1,3)	61.2 (1.2)	0.465 (0.032)	0.586 (0.15)	0.487 (0.05)
PCA (1,3)	**62** (0.5)	**0.547** (0.019)	**0.617** (0.007)	**0.552** (0.016)
PCA (1,4)	61.4 (1.1)	0.545 (0.009)	0.611 (0.007)	**0.552** (0.01)
PCA (1,5)	61.3 (1.2)	0.544 (0.007)	0.61 (0.007)	**0.552** (0.01)

Table 5.3: SVM best results for different n-gram ranges. The number in brackets next to the type denotes the range of n-grams used. All entries have min_df = 10. Non-PCA entries both have $C = 2$. PCA (1,3) has $C = 3$, min_df = 10 and n_components = 250. Both PCA (1, 4) and (1,5) have $C = 2$ and n_components = 400.

too much, minimum DF of 5 and 10 had a similar performance and $0.1 \leq \alpha < 1$ were generally increasing the F_1-Score the closer to 0.1 they got. For the Gaussian NB, the default variation smoothing was 10^{-9}, which was too low. We performed a search on linearly distributed numbers on a logartihmic scale starting with 10^{-9} and ending with 10. The best value was around $4 \cdot 10^{-4}$. Surprisingly, even though the accuracy was much lower than the other versions, the F_1-score was better than in the Multinomial case. PCA was also tried, however, it was really hard to find a configuration that would get even close to the raw TF-IDF.

In most of the cases, replacing the URLs did not improve performance, which is unexpected. One would expect that by using higher thresholds for DF, the hyperlinks would not be included either way. Perhaps there were some URLs that were repeated in multiple posts, and they could have been used as a feature.

The Complement NB is supposed to work well in cases of unbalanced classes, which is definitely our case, and it has proven itself. Naive Bayes in general achieves a good performance even from small amounts of data, which, as we are going to see, hampers other models.

5.2.2 Support Vector Machines

With SVMs, the theory is complicated, but using them is simple. Most of the time, we want to use the RBF kernel. We only have to choose one hyperparameter, C, which is inverse of the regularization strength. Other than that, we will only play with text preprocessing. The only other parameter we will be changing is the tolerance for stopping the support vector optimization. We have changed it from 10^{-3} to 10^{-4}, but usually we converge either way. We have performed experiments with and without PCA, and surprisingly, PCA actually helped in this case. The best configuration results for each case can be found in Table 5.3.

From the results, we see that increasing the range of n-grams used actually helped performance. Naturally, this way the model has more information to work with. Sometimes, it can be too much information, as in the case of Naive Bayes, but SVMs can use it to find better support vectors. The condensed representation extracted from PCA actually helped the performance a lot, surpassing the previous models. PCA is similar to what SVMs do, in the way that it finds the most important eigenvectors, and transform the data into the best approximation in a space with the eigenvectors as a base. SVMs also find important vectors,

which they then use to transform data, project it into a different feature space, and then classify them based on these new features.

Even though the version with n-gram range (1,3) has the best F_1 score, all the top SVM configurations with PCA are very close to each other in all metrics. The best number of components seems to be somewhere between 200 and 500, which, is around $15 - 40\%$ of training examples in our case.

5.2.3 Neural Network Ensemble

In contrast to SVMs, Neural Networks can have quite a lot of hyperparameters to choose and optimize. The architecture of the network is an obvious one, and generally one also thinks about learning rate, batch size and number of epochs to train for. Of course, we can use different activation functions as well, but ReLU is generally the best option, and theoretically optimal. However, we can use some more optimizations to squeeze performance out of our models. Here are a few:

- Label smoothing - For classification, model outputs a distribution, where each position gives a percentage, which says how likely the given class is given the input features. We usually use a one-hot representation of the training labels as the data the model should strive to predict. Because of this, the model will always try to push one class to 1 and the rest to zero, which is almost impossible and leads to overfitting. We could instead smooth the labels, i.e. take some percentage from the actual target class, and distribute it to the others. This gives the model some leeway with which to adjust weights, if it becomes too confident in one class.

- Learning rate decay - During training, it is often beneficial to have a high learning rate at the beginning, because we want to quickly descend the gradient of the loss function. In the later stages, we want to make small steps to fine-tune the weights, i.e. a small learning rate. We can achieve this using learning rate (LR) decay. It is simply a function which, for a given step outputs the learning rate that the model should use. It could be linear, exponential or even cosine.

- Dropout - During training, we can ignore the output of some percentage of neurons on the previous layer. This can help the network create better, more independent features on every neuron, since they cannot rely on all the information being there during the next forward pass. This has some technical consequences for the gradient computation, but they are easily solved. Since dropout effectively decreases the "strength" of the network, by decreasing the average amount of neurons active in any given step, it can also hamper the network.

During training, we can see the effect of PCA quite well. The time to compute one training step without using PCA varies between 50 - 130ms, depending, among other things, on the range of n-grams and size of the network. With PCA, this time is cut down to around 10 - 20ms. However, we may some accuracy by doing this.

Another observation is, that the small size of the dataset means models tend to overfit very early, and achieve high accuracy on the training set (close to 1)

Type	Acc% (σ%)	F_1 (σ)	$F_1^{(w)}$ (σ)	$F_{0.5}$ (σ)
base	58.1 (1.5)	0.466 (0.023)	0.564 (0.016)	0.489 (0.029)
ls = 0.1	57.6 (1.5)	0.46 (0.021)	0.558 (0.016)	0.483 (0.027)
PCA 250	58.1 (1.4)	0.469 (0.029)	**0.565** (0.012)	0.484 (0.039)
PCA + ls	58 (1.7)	**0.47** (0.029)	0.564 (0.016)	**0.484** (0.035)
PCA + ls*	57.5 (2)	0.468 (0.029)	0.561 (0.17)	0.482 (0.036)
patience = 10	**58.7** (0.6)	0.417 (0.014)	0.553 (0.008)	0415 (0.029)

Table 5.4: Neural network optimizations. ls is the label smoothing strength. ls* is label smoothing = 0.3. All models have learning rate = 0.001, Adam optimizer, 5 models in the ensemble, each having resampled training data, one hidden layer with 256 neurons, minimum DF = 10, n-grams of range (1, 3) and cosine decay.

Type	Acc% (σ%)	F_1 (σ)	$F_1^{(w)}$ (σ)	$F_{0.5}$ (σ)
best overall	**57.1** (3)	**0.507** (0.026)	0.568 (0.028)	**0.511** (0.028)
best char	57 (2.3)	0.497 (0.02)	**0.569** (0.022)	0.498 (0.02)

Table 5.5: Nerual networks final results. "best overall" had a number of PCA components = 300, learning rate: 0.001, number of models in the ensemble = 7, hidden layers = [256, 256], resampled data without subsampling, epochs = 30, batch_size = 64, dropout = 0.2, label_smoothing = 0.1, patience = 20. "best char" had a number of PCA components = 400, character-level n-grams with a range of (3,7). Other hyperparameter were the same.

after a few epochs. This is even with very small sizes, such as only 1 layer with 256 neurons. However, using PCA, at least on small network sizes, helps with this problem.

Patience (stopping training after some metric on validation data is not increasing anymore) is usually an important part of training, especially in cases where models tend to overfit. However, sometimes, the results it gives are seemingly worse than without it. For example, in our initial experiments in Table , the last row includes setting patience to a generous 10 epochs (when we train for 20 epochs in total). An optional setting is that we restore the model to the best weights it had during the "activation" of patience, i.e. when the metric we are tracking starts increasing when we want it to decrease etc. In our case, the tracked metric is the validation loss, which is usually the right choice. However, for one reason or another, the validation F_1-Score is increasing even when the loss starts to increase. This may be because the model starts to misclassify some examples in the more represented classes, but correctly classifies examples in the less represented classes. Since the patience mechanism restores the weights when the validation loss was at its lowest point, we get a worse F_1-Score than without it. This does not mean that the patience mechanism is bad, only that caution is always advisable when adding optimizations to one's models.

After some further experimentation with different layer sizes, dropout rates, number of models in the ensemble and character-level n-grams, we have arrived at the results in Table 5.5.

Overall, it was beneficial to use 2 hidden layers with a mild dropout, and to expand the word n-gram range to (1,4). The model had more "memory" to work

with, which likely improved results. The best model with character level ngrams used a range of (3, 6), whose upper bound is close to the length of one word. During training, there were some models that had validation F_1-Scores around 0.55, however, many of them had below average F_1-Score too. This indicates that there is potential improvement to be gained from using character n-grams, but we were unable to achieve it.

Neural networks did not achieve as good performance as SVMs or Naive Bayes models, likely because the amount of data was too small and too unbalanced.

5.2.4 RNNs

Recurrent Neural Networks are a very popular, and currently the state-of-the-art choice for language processing. This makes them a great choice for sentiment analysis. However, there is one problem: we do not have enough data. RNNs are unexpectedly deep, in the sense that they have many hidden layers, all with their own trainable parameters. There is one RNN cell for every token in the input sequence, and this cell can include multiple weight matrices. This makes them very prone to overfitting, when working with a small amount of data. And this is exactly what happened in our case.

Another problem is that usually, we have to use fixed-length sequences on the input. TensorFlow gives us access to something called Ragged tensors, which are basically variable-length tensors. They are compatible with their implementation of RNNs, which solves this problem. However, during training, it seems that TensorFlow finds the maximum length of the ragged tensors from training data, and "pads" all other tensors to this length. It is unclear whether it just creates placeholder RNN cells for every potential token, or takes up more VRAM differently, but at some point, there is not enough VRAM on the graphics card used for these experiments (when a token is a single character especially). So we are slightly limited in the experiments we are able to perform.

One measure we tried to combat this is to "shorten" some posts. There are a few posts in the training dataset that are much longer than 1000 characters, which cause these issues. In these posts, GameStop is usually mentioned only once or twice in passing, in the middle of a diatribe about something unrelated. That is why we have tried picking out only some "windows" around these occurrences in a given post, and returning a union of them. A window of size k consists of k tokens before the given token, and k tokens after. A token can be a whole word or a single Unicode character. We can set a maximum posts length in tokens, above which we apply this shortening procedure. These overlapping windows should provide a good approximation of the post, and it makes training both possible and much faster.

During experiments, an interesting problem has happened. If we set learning rate too low, for example to 0.0001, we may become stuck in a local optimum where it seems that nothing at all changes for around 10 or so iterations. The network seems to assign one label to short posts or tweets, and only makes real decisions on the longer posts. This is likely caused by the huge disparity between the length of Reddit posts and Tweets. Experiment results can be found in Table 5.6

The results are very noisy. Some iterations were great, with F_1-Scores in the

Type	Acc% (σ%)	F_1 (σ)	$F_1^{(w)}$ (σ)	$F_{0.5}$ (σ)
LSTM single	49.2 (6.0)	0.483 (0.062)	**0.405** (0.064)	**0.419** (0.068)
GRU single	45.7 (6.2)	0.456 (0.055)	0.374 (0.057)	0.385 (0.059)
LSTM double	48.1 (4.6)	0.473 (0.48)	0.384 (0.047)	0.4 (0.057)
LSTM double*	**49.9** (6.2)	**0.487** (0.058)	0.388 (0.049)	0.396 (0.054)

Table 5.6: RNN results. All models had a learning rate = 0.001, batch size = 32, label smoothing = 0.1, max vocabulary size = 7000, an embedding layer with output size of 32, patience = 20 and were trained for 30 epochs. The single RNN layer variants had their output (and hidden state) sizes of 128. Double layer versions had the same output size. The "double*" variants first hidden layer had an output size = 256.

Type	Acc% (σ%)	F_1 (σ)	$F_1^{(w)}$ (σ)	$F_{0.5}$ (σ)
min_df = 2	60.9 (3.3)	0.421 (0.024)	0.567 (0.032)	0.418 (0.026)
min_df = 3	**61.4** (2.4)	**0.425** (0.018)	**0.573** (0.023)	**0.422** (0.18)
min_df = 5	60.7 (1.7)	0.421 (0.01)	0.567 (0.013)	0.416 (0.011)
min_df = 10	59.6 (1.5)	0.414 (0.011)	0.557 (0.014)	0.407 (0.11)
ngrams = (1,2)	60.2 (2.2)	0.416 (0.017)	0.561 (0.022)	0.412 (0.018)
ngrams = (1,4)	60.6 (2.7)	0.419 (0.02)	0.565 (0.027)	0.417 (0.021)

Table 5.7: Random forest preprocessing results table. min_df denotes the minimum document frequency of a token in the vocabulary. All models have been training with Gini impurity, 100 trees, min_df = 2 and (1,3) n-grams, if not stated otherwise.

0.54, and others were abysmal. There is definitely potential for good performance, but the network seems to overfit to the longer posts, or perhaps posts of one class, as we can see in the small weighted F_1-score. Overall, we were unable to make RNNs match the performance of other models on this dataset.

5.2.5 Random Forests

Finally, we have our last type of model, the Random Forests. For hyperparameters, we can mostly control how are the different trees built and how many of them will be there in the ensemble. We also have AdaBoost and Gradient boosting versions, which will (hopefully) improve performance. Results can be found in Table 5.7.

First we set the hyperparameters to a general standard: (1,3) n-grams, 100 trees, Gini impurity as the criterion and to use at most $\sqrt{D}$ features when deciding to make a split. D is the length of a single feature vector. First experiments were to determine the best preprocessing hyperparameters. It seems that unlike the other models, random forest do better if they have even less common features to choose from, i.e. they worked best with min_df = 3. Then, we proceeded with trying different options for model parameters.

Simply increasing models surprisingly did not improve performance, but when combined with increasing the maximum amount of possible features to choose from during node splitting, we have made some improvements. Entropy was also tried as a possible splitting criterion, and it had similar performance to

Type	Acc% (σ%)	F_1 (σ)	$F_1^{(w)}$ (σ)	$F_{0.5}$ (σ)
n_models=200	60.6 (0.02)	0.419 (0.015)	0.565 (0.02)	0.415 (0.016)
max_feats=0.3	61.1 (2.7)	0.434 (0.021)	0.576 (0.025)	0.433 (0.027)
max_feats=0.3	61.1 (2.7)	0.434 (0.021)	0.576 (0.025)	0.433 (0.027)
max_feats=0.4	61.1 (2.1)	0.444 (0.028)	0.578 (0.024)	**0.455** (0.045)
max_feats=0.6	**61.4** (2.3)	**0.446** (0.023)	**0.582** (0.023)	**0.455** (0.034)
msl = 3	60.2 (3.0)	0.431 (0.025)	0.568 (0.029)	0.434 (0.033)
entropy	60.7 (2.4)	0.44 (0.023)	0.574 (0.024)	0.451 (0.03)
PCA	60.9 (0.02)	0.424 (0.015)	0.571 (0.019)	0.416 (0.014)
ccpa = 0.001	0.613 (2.3)	0.446 (0.023)	0.581 (0.022)	**0.455** (0.034)

Table 5.8: Random Forest hyperparameter results. Unless stated otherwise, all models have been trained with n_models = 200, min_df = 3, ngrams = (1,3), minimum samples per leaf (msl) = 1. First row was also performed with maximum features = $\sqrt{D}$. Experiment with msl = 3 was performed with maximum features = 0.4. Experiment with entropy was performed with maximum features = 0.3 and msl = 3. The ccp row means cost-complexity-pruning alpha, and it had max_features = 0.6, and it's F_1-Score was strictly smaller than the highlighted maximum. PCA had msl = 2, min_df = 2, and used 300 components. Other values for each hyperparameter were tried, here are only the best performing ones, or ones used for demonstration purposes.

Gini impurity. Setting minimum examples for creating a leaf did not improve performance (when a model already had a higher maximum number of features). Cost-complexity pruning was also tried, but it did not achieve better performance either. It did increase training time by a factor of 2. PCA was also tried, and it had similar results to every other method, but did not improve upon any configuration tried.

Finally, we have tried using AdaBoost and Gradient boosting to improve performance. These methods are computationally difficult, since they are difficult to parallelize, and, in the case of gradient boosting, have to be done sequentially. Both can easily increase training time by a factor of 10 or more. PCA immensely helps with reducing this training time, and is almost required, considering the dimensionality of our data. For gradient boosting, we use a different node splitting criterion, called Friedman MSE. We will not be going into detail on how it works, but it is better suited for gradient boosting, according to [40].

After trying various learning rates, it seems that around 0.01 seems to be good for AdaBoost and around 0.5 - 0.7 for gradient boosting. This high learning rate is not surprising, since we want to make big steps creating new trees and fixing errors.

Overall, AdaBoost was the best variant, with F_1-Score nearing Neural Networks, and with a higher accuracy too. But it is still behind Naive Bayes and SVMs.

Type	Acc% (σ%)	F_1 (σ)	$F_1^{(w)}$ (σ)	$F_{0.5}$ (σ)
AdaBoost	**59.9** (0.02)	**0.509** (0.02)	**0.591** (0.019)	**0.518** (0.018)
GradBoosting	59.4 (1.6)	0.435 (0.027)	0.565 (0.018)	0.4407 (0.046)

Table 5.9: Random forest optimizations table. AdaBoost had min_df = 2, ngrams = (1,3), number of PCA components = 300, learning rate = 0.01, number of models = 200, minimum samples per leaf = 3, and maximum number of features considered for splitting = 0.3. Gradient Boosting had a learning rate = 0.7, criterion was Friedman MSE, and otherwise the same as AdaBoost

Type	Acc% (σ%)	F_1 (σ)	$F_1^{(w)}$ (σ)	$F_{0.5}$ (σ)
SVM	**62** (0.5)	**0.547** (1.9)	**0.617** (0.006)	**0.552** (0.016)
ComplementNB	61.7 (1.7)	0.53 (0.015)	0.612 (0.017)	0.535 (0.015)
AdaBoost	59.9 (2.0)	0.509 (0.02)	0.591 (0.019)	0.518 (0.018)
NN	57.1 (3.0)	0.507 (0.026)	0.568 (0.028)	0.511 (0.028)
LSTM	49.9 (6.2)	0.487 (0.058)	0.388 (0.049)	0.396 (0.054)

Table 5.10: Best model results. Table contains results of the best performing (w.r.t. F_1-Score) model from each model type. The best SVM configuration was $C = 3$, minimum document frequency = 10, word-level n-grams with a range of $(1, 3)$ and PCA with 250 components. Results for other models can be found in their respective sections, in the result table descriptions.

5.2.6 Result summary

In Table 5.10 we can find the best models from each category. SVMs are a clear winner in all categories. Surprisingly, Naive Bayes is also of similar performance. The results are not completely surprising given the small size and unbalanced nature of the dataset. Neural networks struggle when not presented with enough data, especially RNNs. In their case, another problem was the large disparity between the length of different posts in the training data. On the other hand, Naive Bayes is well suited to small dataset, and the Complement version of Multinomial Naive Bayes is also very well suited to unbalanced data. Random Forests could potentially be used too, but they are simply middle of the pack with generally average performance, with one exception being the AdaBoost variant. If not for AdaBoost, they would be on average trailing behind other models. Finally, SVMs are the overall best-performing thanks to their synergy with PCA.

However, even the best accuracy and F_1-scores are relatively small. It is questionable whether the performance of models on this dataset could be improved by much. As stated before, there is simply not enough data. The unbalanced nature of data would not be such a hindrance if we had thousands or tens of thousands of posts to train on. Because there was only one person annotating, there was simply not enough time to create a large training set. Nonetheless, these models are much better than simply guessing randomly, and so we can still get meaningful results.

We have also rated the models according to their performance on our data in Table 5.11. The ratings are in three categories: Training speed, performance on small datasets, and performance on unbalanced data.

Naive Bayes models are in general very quick and easy to train, making them

Model	Training speed	Small dataset	Unbalanced data
Naive Bayes	+ + +	+ + +	+ + +
SVM	+ +	+ + +	+ +
NN	+	- -	+
RNN	- - -	- - -	-
Random Forest	- - -	+ +	- -

Table 5.11: Model rating according to their performance on data used.

a good baseline. They achieve a good performance on small datasets, and can also handle unbalanced data when the correct model variant is used.

Support vector machines are relatively quick to train (certainly compared to the rest of models used), and have no problems with small datasets. From the results, we can deduce that they can also handle unbalanced data. One caveat is, that SVMs are best suited to binary classification. For multi-class classification, either a one-versus-all or one-versus-one scheme has to be used, which can lead to inaccuracies.

Simple neural networks can take a long time to train if we use large hidden layer sizes or complex activation functions. They can also overfit easily, thus needing large training datasets. If given a large training set, they can handle unbalanced data.

Recurrent neural networks take a very long time to train, which is entirely dependent on the length of individual input sequences. If we have large disparities between the lengths of examples, they can very easily overfit on the longer sequences, and ignore the short ones. They are not a good option for small datasets. For unbalanced data, they are similar to simple neural networks, but could have problems if the length of examples in less represented classes is shorter than the over-represented ones.

Finally, random forests are potentially very slow to train, especially if we have very high-dimensional data. Especially the gradient boosting variant is hard to parallelize. On the other hand, if given low-dimensional small datasets, their performance is not bad. Unbalanced data can potentially be a problem because of the simplicity of the base decision trees. Often, we may have a situation where the limit on minimum examples per leaf might lead to misclassification of the less represented class. If we do not set a minimum amount of examples per leaf, the tree may heavily overfit to the training set.

5.2.7 Sentiment estimation

We will now use the best model from the previous section, the SVM, to calculate the sentiment of the whole corpus of Tweets and Reddit posts that we have gathered, and plot it in a graph. The sentiment was calculated to include the "score" of a post in a given day. For Reddit posts, the score is *number of upvotes*$\cdot$ *upvote ratio*, and for Tweets it is simply the Like count. The sentiment score is on a scale of -1 to 1, and is calculated as follows

$$\text{sentiment score} = \frac{\text{sum of positive scores} - \text{sum of negative scores}}{\text{sum of all scores}}$$

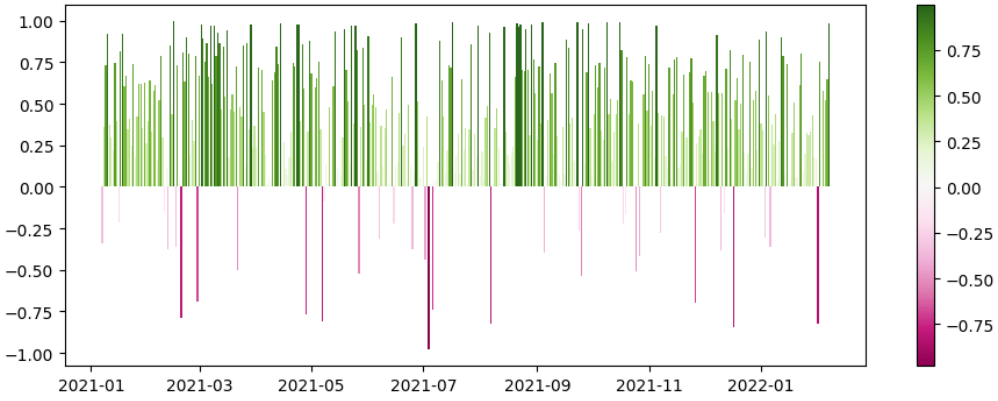


Figure 5.2: GameStop sentiment over the year.

From the graph we can see that the sentiment is overwhelmingly positive all year, as expected, with onlt a few negative days. The most negative day was the 4th of July. This day, together with 2nd and 6th of July that were negative, seem to have been around a drop in GameStop's share value and news about the upcoming stock split. Another quite negative day was the 19th February 2021, this seems to have been the day when GameStop's price bottomed out after the short squeeze a few weeks earlier. The other negative days seem to generally be around the dips in the stocks price, like the one in February 2022.

Similarly, we have exceptionally strong positive sentiment around the peaks of the stocks price, for example in April and by the end of August, both of which were relatively big rallies.

Overall, we can conclude that the first hypothesis of 4.2.1 was correct. Predictions seem to follow the expected sentiment when looking at the stock price during the year. Given the relatively unfavorable model performance, this is a better result than expected.

5.3 Node importance estimation results

In this section we will discuss the results of node importance estimation according to the metrics mentioned in Section 2.4. HITS scores have been calculated for up to 100 iterations, with an error tolerance of 10^{-8} when checking for convergence. PageRank was estimated with $\alpha = 0.85$ and error tolerance of 10^{-6}, also for up to 100 iterations. In both cases, algorithms converged before the iteration limit was reached.

5.3.1 Twitter

The results are split into two tables. In table 5.12 we can see top 10 users by PageRank and HITS Authority score. As expected in the second hypothesis in Section 4.2.1, GameStop's Twitter account is the highest ranking account by both of these metrics. Ryan Cohen is also quite high in terms of both PageRank and Authority score. He is in the top 5 in both cases, but we would have expected

Pos	by PageRank (score)	by Authority Score (score)
1.	GameStop (0.032093)	GameStop (0.035445)
2.	[User 2] (0.019764)	[User 6] (0.028672)
3.	[User 3] (0.017900)	[Unk 331] (0.022234)
4.	Will Meade (0.015784)	Ryan Cohen (0.020616)
5.	Ryan Cohen (0.011425)	Keith Gill (0.019902)
6.	Liz Claman (0.010991)	[User 7] (0.018408)
7.	[User 4] (0.010688)	[User 8] (0.018379)
8.	Dan Carney (0.010106)	[User 9] (0.005415)
9.	Frank Nez (0.008175)	[User 10] (0.018185)
10.	[User 5] (0.007819)	[User 11] (0.018181)

Table 5.12: Twitter node metric table 1. Names surrounded by square brackets are anonymized to protect the identity of those users. Only public figures are named. User [Unk 331]'s Twitter profile was deleted, so their identity is unknown.

him to be in the top 3. Nonetheless, his influence on the GameStop discourse is rightly confirmed by these results.

Keith Gill, being one of the most important people in the GameStop debacle, is also in the top 5, for the Authority Score. Surprisingly, he is not in the Top 10 by PageRank. His position in the community lends itself well to having a high authority score - people often refer to his initial Due Diligence posts explaining his reasoning about GameStop's hidden value. However, it seems that his accounts' inactivity has led to a lower PageRank score.

Other named users in this table are Liz Claman - a business news anchorwoman, Dan Carney - a comedian and Frank Nez - a finance blogger and influencer. Except for Dan Carney, it is not surprising that these people are ranked as one the most important according to PageRank, because of their reach and interest in these topics. The anonymized users are mostly minor financial influencers and speculative investors.

Pos	by Hub Score (score)	by Betweenness Centrality (score)
1.	[User 12] (0.095917)	[Unk 331] (0.002300)
2.	[Unk 226] (0.014751)	[Unk 226] (0.001424)
3.	[User 13] (0.010923)	[User 6] (0.001339)
4.	[User 14] (0.010520)	[Unk 270] (0.000926)
5.	[User 6] (0.010447)	[User 19] (0.000871)
6.	[Unk 371] (0.009197)	[User 20] (0.000742)
7.	[User 15] (0.008772)	[User 21] (0.000717)
8.	[User 16] (0.008618)	[User 22] (0.000702)
9.	[User 17] (0.008384)	[User 23] (0.000639)
10.	[User 18] (0.008359)	[User 24] (0.000580)

Table 5.13: Twitter node metric table 2. Anonymization same as in Table 5.12

.

In Table 5.13, we see the results for Hub score and Betweenness centrality estimation. Because of anonymization, this table does not give us too much

Pos	by PageRank (score)	by Authority Score (score)
1.	klay (0.007767)	richard.shapiro (0.020522)
2.	tana.jones (0.003896)	james.steffes (0.018358)
3.	sara.shackleton (0.003699)	paul.kaufman (0.018340)
4.	ebass (0.003650)	susan.mara (0.017643)
5.	jeff.skilling (0.003107)	karen.denne (0.016001)
6.	kenneth.lay (0.002910)	skean (0.014101)
7.	gerald.nemec (0.002745)	sandra.mccubin (0.013765)
8.	mark.taylor (0.002687)	harry.kingerski (0.013269)
9.	louise.kitchen (0.002562)	james.wright (0.012421)
10	jeff.dasovich (0.002296)	mpalmer (0.011701)

Table 5.14: Enron node metric table 2. Emails are displayed without the part after "@", since for all of them it is "@enron.com". Betweenness centrality values are normalized by $1/((n-1)(n-2))$

information at the first glance. Surprisingly, none of the public accounts in the top 10 ranks of PageRank and Authority scores are in this list.

After going through these profiles, they are mostly minor financial or crypto influencers, or ordinary people. Their followers number in hundreds to small thousands, and they themselves follow hundreds to small thousands of profiles. Interestingly, a large part of the influencer profiles have been created at the beginning of 2021, specifically around April.

We can conclude from this that Hubs, which were defined as nodes in the network amassing links to good authorities, are in this case minor actors. Similarly, in the case of Betweenness centrality, these people are the bridges connecting small communities together.

The absence of major public accounts in top positions of Hub Score and Betweenness centrality score lists has one likely explanation. The major public accounts do not usually link to every other major account, they just post Tweets. These people on the other hand likely retweet many posts by the major public accounts, in addition to tweeting themselves, which increases their Hub score.

All in all, the second hypothesis is only half-correct. The mentioned accounts were important, but only the PageRank and Authority scores.

5.3.2 Enron

For Enron, the Betweenness centrality calculation would be too time-consuming because of the size of the network. Instead, we sample a fraction of nodes, called "pivots", and use them to estimate the total betweenness. This method, by U. Brandes and C. Pich, is described in [41]. We have chosen $k = \lfloor n/2 \rfloor$, nodes randomly for this purpose, n being the number of nodes in the graph.

From the results in Table 5.14 we see that one of Kenneth Lay's emails dominates the PageRank ranking, with other higher ups at Enron filling out most of the ladder. People like Jeff Skilling (CEO of Enron) Sara Shackeleton (VP of Enron), Louise Kitchen (COO of Enron Americas) are in the list, which is not at all surprising.

However, we also have tana.jones, whose real identity seems to be unknown.

Pos	by Hub Score (score)	by Betweenness Centrality (score)
1.	jeff.dasovich (0.334277)	jeff.dasovich (0.007732)
2.	susan.mara (0.130143)	jeff.skilling (0.005936)
3.	ginger.dernehl (0.038967)	j.kaminski (0.005932)
4.	mary.hain (0.026147)	louise.kitchen (0.005303)
5.	james.steffes (0.022168)	m..presto (0.004833)
6.	miyung.buster (0.020235)	kenneth.lay (0.004140)
7.	sgovenar (0.015922)	sally.beck (0.003949)
8.	alan.comnes (0.014196)	gerald.nemec (0.003768)
9.	christi.nicolay (0.012599)	40enron(0.003582)
10	rhonda.denton (0.011536)	sara.shackleton (0.003358)

Table 5.15: Enron node metric table 2. Same formatting as in Table 5.14.

She seems to have been an important in the internal workings of Enron. By looking at related emails, it seems that she may have been an accountant or perhaps was managing some financial contracts with other companies.

Person with the highest Authority score was Richard Shapiro, a senior VP, which is not surprising. In fact, the whole list is made up of similarly high ranking people at Enron. This is very similar to the Twitter network, where high PageRank and Authority score people were usually public figures, not ordinary people.

In case of Hub score, the positions of people in the top 10 changes a bit. At the top we have two big hubs: Jeff Dasovich, government relations executive at Enron, and Susan Mara, director of California regulatory affairs at Enron. It makes sense that these people would receive and send many emails to both their subordinates and high ranking officers at Enron. However, the other people are not necessarily that high in the corporate structure. For example, we have Mary Hain, a government affairs lawyer at Enron. Interestingly, most of the people on this list are somehow related to government or regulatory affairs.

For the Betweenness centrality score, we once again see many executives and directors dominate the list. This is in contracts to the Twitter network, in which both Hub scores and Betweenness centrality scores have been dominated by mostly ordinary people and minor influencers. However, in both networks, people on this list act as bridges between communities.

These results seem to generally support the second hypothesis in Section 4.2.2. The corporate structure of Enron means that people in teams communicate with their supervisors, they in turn communicate with their supervisors and so on, until we reach the top level executives. Betweenness score is calculated based on the fraction of the shortest paths between nodes going through a given node, and this tree-like corporate structure leads to high betweenness scores for managerial roles. In the aforementioned hypothesis, we theorized that middle-management roles would have high betweenness, and we see upper-middle management and some executives in the top ranks of betweenness. This does make sense, since the upper echelons of the company act as relays for a potentially larger fraction of nodes.

Degree type	$\hat{\gamma}$	$\sigma_{\hat{\gamma}}$	$\hat{k}_{min}$	$\sigma_{\hat{k}_{min}}$	p
in	1.8483	0.0274	3	0.6957	0.8366
out	2.3766	0.0829	2	1.2472	0.0022
total (all)	2.1476	0.0202814	2	0.4995	0.0294

Table 5.16: Twitter parameter estimation table. Rows are for different degree distributions of the network, i.e. "out" means we consider the degree distribution of node out-degrees. Nodes which have degree values of 0 are pruned from the data prior to estimation. $\hat{\gamma}$ is the estimated scaling exponent of the fitted power law, $\sigma_{\hat{\gamma}}$ is the standard deviation. Similar notation for k_{min}. In the last column we have the p-value.

5.3.3 Result summary

Twitter

For the Twitter network, we found out that public figures have the highest PageRank and Authority scores as expected, with GameStop being the highest ranking in both scores. Conversely, small influencers and ordinary users dominate regarding Betweenness and Hub scores, because they likely retweet public figures, other influencers, and also post themselves. This is what makes their Hub scores high. They usually have a small following, and similarly follow many accounts, which likely makes them act as bridges between communities in the network, and thus they also have a high Betweenness centrality.

Enron

This network's top PageRank and Authority scores are populated by top level executives and officers, which is a similar result as in the case of Twitter. On the other hand, Hub and Betweenness centrality scores are also populated by senior staff, but for similar reasons as in the case of Twitter. This is because the corporate tree-like structure naturally makes management both hubs and relays of information between different parts of the company.

5.4 Power-law fit investigation results

Following the steps in Section 2.3.1, we have obtained the following results.

5.4.1 Twitter

For twitter, we have used $n = 5000$ bootstrap simulations for both standard deviation estimates and p value calculation. The parameter estimation begins with a rough MLE estimate given by The results can be found in Table 5.16. We can rule out the power law for both the in- and out-degree distributions, as their p value is less than the threshold of 0.1. The fact that the power law is not a good fit for the out- and total degree distributions is surprising.

It can likely be explained by the relatively small amount of data available compared to the actual activity on Twitter. This is because we did not have access

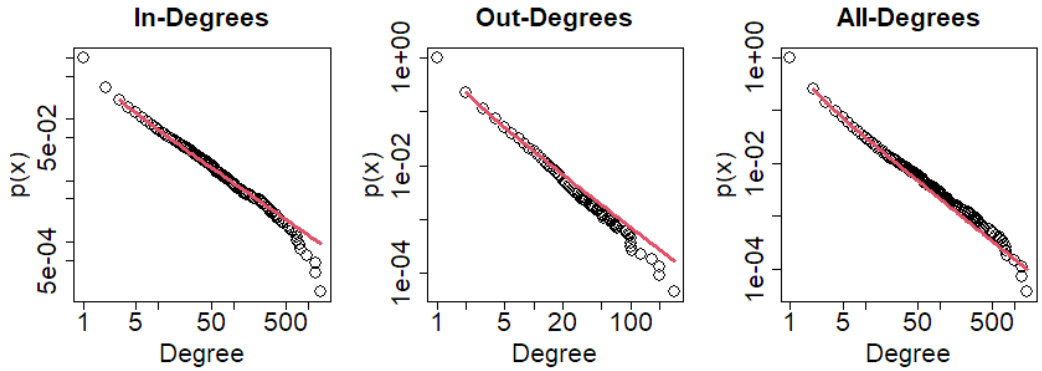


Figure 5.3: Twitter degree distributions. Logarithmic binning applied to the x-axis, both axes are in log-scale. The red line represents the best fit power law model for the given distribution, beginning at k_{min}.

Degree type	$\hat{\gamma}$	$\sigma_{\hat{\gamma}}$	$\hat{k}_{min}$	$\sigma_{\hat{k}_{min}}$	p
in	2.8643	0.3318	53	17.7980	0.496
out	2.9948	0.3307	196	62.9954	0.558
total (all)	1.7069	0.4538	3	70.0855	0

Table 5.17: Enron parameter estimation table. Same notation as in table 5.16

to data provided directly by Twitter, but only a third-party scraping service, which does not guarantee coverage of all tweets posted.

On the other hand, we cannot rule out a power law for the in-degrees. Interestingly, the estimated scaling parameter is outside the expected range of $\gamma \in [2, 3]$, and also with a small standard deviation. From Figure 5.3 we can see that for the power law model consistently over-estimates the degree probabilities in the empirical distribution. Conversely, in the total degree distribution, the model under-estimates the degree probabilities.

5.4.2 Enron

Enron is a much larger graph in terms of the number of nodes and edges, so we have only used $n = 1000$ simulations for bootstrapping. The results are in Table 5.17.

From the results we can see that the power law is a plausible fit for the distribution of in- and out-degrees, with p values higher than our threshold. However, we see large deviations in the estimated k_{min} in both cases. To see why, we refer to the histogram of estimated k_{min} observed during bootstrapping in Figure 5.4. In the case of out-degrees, there are peaks around degree 60 as well as a clustering of values around 200, so it is possible that the real k_{min} could be around 60.

The distribution of k_{min} values is even more polarized for total degrees. Vast majority of them are close to the estimated value of 3, but there are some experiments with k_{min} values between 40 and 220, which heavily increases the standard deviation.

In both cases however, the estimated k_{min} is quite large, which means we have to discard a large part of data to get a good fit to the power-law, so it is likely

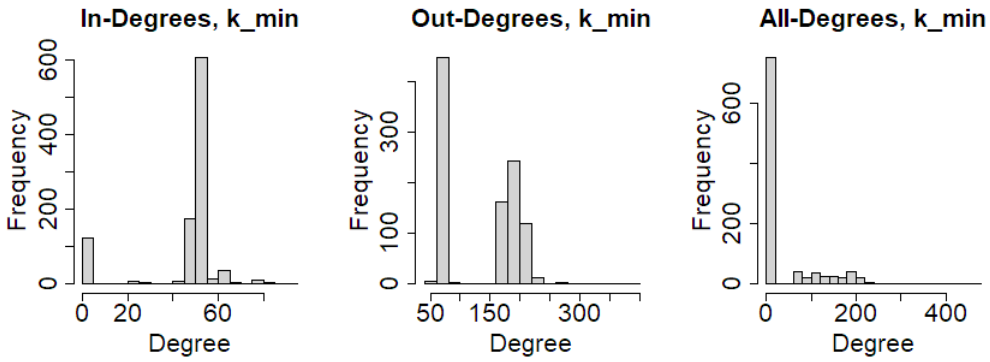


Figure 5.4: Histogram of estimated k_{min} parameters during bootstrapping for the Enron network.

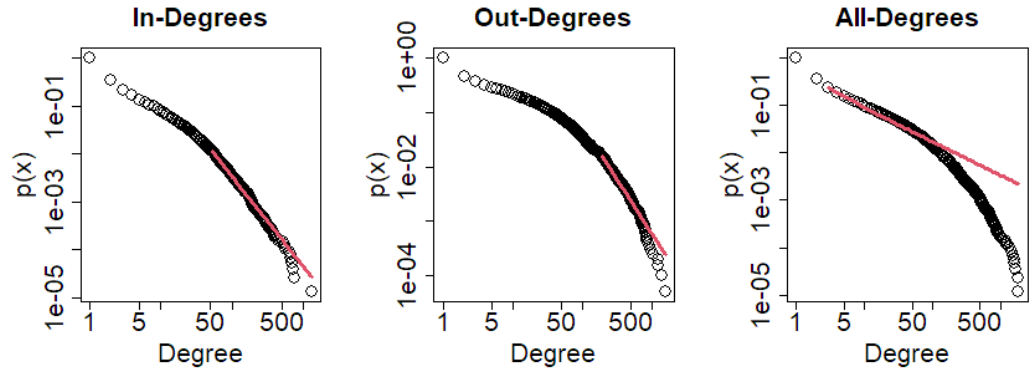


Figure 5.5: Enron degree distributions. Same scaling as in Figure 5.3.

that another distribution may be a better fit.

We can also see that the best-fit power completely misses the tail of the distribution, so it is not surprising that the p-value is 0.

5.4.3 Result summary

For the Twitter network, we cannot rule-out the power-law only in the case of the in-degree distribution. Both out- and total degree distributions have p-values close to zero, and thus the power-law is a bad fit.

For the Enron network, we can rule out the power-law in the case of the total degree distribution. While, in- and out- degree distributions fit relatively well into a power-law, another distribution may be a better fit. This is because of the high value of the estimated lower bounds on the power-law behavior, which means we have to throw

Conclusion

In this thesis, we aimed to investigate the sentiment within the GameStop community on Twitter and the subreddit r/wallstreetbets, w.r.t. GameStop post price spike. We also wanted to identify influential actors in this community using data collected from Twitter. We have addressed this objective by constructing a social network from the Tweets and the subsequent application of various node importance measures discussed in the theoretical section. We also performed this analysis on the Enron email dataset, from which we also created a social network. Additionally, we wanted to see if the degrees of these two social networks follow a power-law distribution. We fitted this distribution to the networks and evaluated the plausibility of our fit via the goodness-of-fit test.

We trained various classifier models for sentiment analysis on manually annotated data gathered from r/wallstreetbets and Twitter. Based on the obtained results, we concluded that the best-performing model was the Support Vector Machine (SVM). The resulting average accuracy of 62% and macro averaged F1-score of 0.547 was not very high because of the dataset's small size and unbalanced nature. The predicted sentiment was overwhelmingly positive, as expected. However, some negative days seemed to correlate with GameStop stock value drops or negative news.

Our research on node importance estimation of the Twitter and Enron networks led to exciting insights into the structure of these networks. Public figures and high-ranking officials predominantly occupied the top ranks in PageRank and HITS Authority scores. However, the Hub and Betweenness scores exhibited a different pattern. In the case of Enron, higher-ups in the organization still dominated, but they included staff management officers and government relations positions instead. In the case of Twitter, minor financial influencers and ordinary people obtained higher scores. Their common characteristic was that they all connected small communities of people in their networks, which naturally gave them high betweenness centrality and HITS Hub Score.

When assessing the power-law fit, we found that this type of distribution is a plausible fit for the in-degree distribution of the Twitter network and both in- and out-degree distributions of the Enron network. However, another distribution might be a better fit in the latter case, although we cannot statistically rule out the power-law distribution.

5.5 Future work

Within the framework of our further research, we would like to expand the annotated dataset to support far better use of more advanced state-of-the-art Machine Learning models from Natural Language Processing. Their application would hopefully improve performance during sentiment prediction. For example, an often-used preprocessing step includes creating a word or document-level embeddings using BERT, a transformer model trained on a large corpus of textual data from Google. We can obtain an embedding for the whole document or every token in the input sequence. We could use such an embedding instead of TF-IDF coefficients or for the replacement of newly created embeddings in the case of

recurrent neural networks (RNNs).

Another potential improvement could involve the modification of the sentiment score itself. Now, the criterion only scores the sentiment w.r.t. the total score of posts in a given day. For example, if there were two posts, each with two upvotes or likes and both classified as positive, the sentiment score for that day would be one. However, it would make sense to let the sentiment score also consider the total score observed in the previous days or weeks. A possible solution could be to employ an exponential moving average (EMA) of all the scores for the last n days and use it as the denominator in the sentiment score calculation.

Bibliography

[1] Geoffrey Grimmett and Dominic Welsh. *Probability: an introduction*. Oxford University Press, 2014.

[2] David A Schum. *The evidential foundations of probabilistic reasoning*, page 49. Northwestern University Press, 2001.

[3] Charu C. Aggarwal. *Data Mining: The Textbook*. Springer, 2015.

[4] Albert-László Barabási and Márton Pósfai. *Network science*. Cambridge University Press, Cambridge, 2016.

[5] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Diameter of the world-wide web. *Nature*, 401(6749):130–131, sep 1999.

[6] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, nov 2009.

[7] H. L. SEAL. The maximum likelihood fitting of the discrete pareto law. *Journal of the Institute of Actuaries (1886-1994)*, 78(1):115–121, 1952.

[8] Morris Goldstein, S. Morris, and G. Yen. Problems with fitting to the power-law distribution. *The European Physical Journal B: Condensed Matter and Complex Systems*, 41(2):255–258, 2004.

[9] Heiko Bauke. Parameter estimation for power-law distributions by maximum likelihood methods. *The European Physical Journal B*, 58:167–173, 2007.

[10] Aaron Clauset, Maxwell Young, and Kristian Skrede Gleditsch. On the frequency of severe terrorist events. *The Journal of Conflict Resolution*, 51(1):58–87, 2007.

[11] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Number 57 in Monographs on Statistics and Applied Probability. Chapman & Hall/CRC, Boca Raton, Florida, USA, 1993.

[12] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[13] Geoffrey Grimmett and Dominic Welsh. *Probability: an introduction*, page 14. Oxford University Press, 2014.

[14] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[15] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[17] Christopher Olah. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed: 2023-01-26.

[18] Kaushik Mani. GRU and LSTMs. https://towardsdatascience.com/grus-and-lstm-s-741709a9b9b1, 2019.

[19] Christopher Olah. Understanding LSTMs. http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM2-notation.png.

[20] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches, 2014.

[21] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *CoRR*, abs/1503.04069, 2015.

[22] Christopher Olah. Understanding LSTMs. http://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-var-GRU.png.

[23] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. *Classification and regression trees*. Routledge, 2017.

[24] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.

[25] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016.

[26] Yoav Freund and Robert E. Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In Paul Vitányi, editor, *Computational Learning Theory*, pages 23–37, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[27] Trevor Hastie, Saharon Rosset, Ji Zhu, and Hui Zou. Multi-class adaboost. *Statistics and its Interface*, 2(3):349–360, 2009.

[28] Wikimedia Commons user Lahrman. https://en.wikipedia.org/wiki/Support-vector_machine#/media/File:SVM_margin.png.

[29] Gerard Salton and Chris Buckley. Term weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):323–328, 1988.

[30] Jonathon Shlens. A tutorial on principal component analysis. *CoRR*, abs/1404.1100, 2014.

[31] snscrape main repository. https://github.com/JustAnotherArchivist/snscrape. Accessed: 2023-01-26.

[32] Psaw (pushshift.io api) repository. https://github.com/pushshift/api. Accessed: 2023-01-26.

[33] The gamestop stock frenzy, explained. https://www.vox.com/the-goods/22249458/gamestop-stock-wallstreetbets-reddit-citron/. Accessed: 2023-01-26.

[34] pandas dataframe documentation. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html. Accessed: 2023-01-26.

[35] Enron email dataset. https://www.cs.cmu.edu/~enron/. Accessed: 2023-01-26.

[36] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. Accessed: 2023-04-16.

[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[38] Colin S. Gillespie. Fitting heavy tailed distributions: The poweRlaw package. *Journal of Statistical Software*, 64(2):1–16, 2015.

[39] Jason Rennie, Lawrence Shih, Jaime Teevan, and David Karger. Tackling the poor assumptions of naive bayes text classifiers. *Proceedings of the Twentieth International Conference on Machine Learning*, 41, 07 2003.

[40] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232, 2001.

[41] Ulrik Brandes and Christian Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, 2007.

List of Figures

List of Tables

A. Attachments

A.1 Short documentation

This section contains information about the scripts used in the thesis, from training models to data visualization.

A.1.1 Folder structure

Scripts that handle data gathering, data processing, and so on are placed in the root folder, along with a `README.md` file, that also has a short description of every script. There are 5 other folders, each containing the following:

- **Data** - Downloaded data from Reddit (`reddit.json`) and Twitter (`twitter.json`). Also contains every dataset generated from the data, whose description is in Subsection A.1.6, along with every annotated dataset variation, described in Subsection A.1.4. Enron data is not included, since it is publicly available at `https://www.cs.cmu.edu/~enron/`.
- **Models** - Scripts used to train models. Their description are in Subsection A.1.2.
- **Graphs** - Generated dataset visualizations and graphs used in the thesis
- **Logs** - Custom logs of experiments in `results.json` and tensorboard logs of some RNN experiments.
- **Networks** - Various formats of network generated from Enron and Twitter data. All networks are saved as edgelists, with edge weights included. Twitter has two network versions: one with user IDs as nodes (`twitter_network.csv`), and one with IDs resolved to usernames (`twitter_network_names.csv`).

  There are also JSON files with metrics calculated for both networks, indexed by node labels (`enron_network_metrics.json` and `twitter_network_metrics.json`). There are also compressed versions of both networks, whose filenames end with `.gz`.

A.1.2 Model scripts

Scrips are located in the Models folder. Each one contains functions that handle preprocessing, training and saving of their specific models. Every script also contains a function called `training_wrapper()`, that represents a unified interface for training. We can call this function from other files and pass it model and preprocessing arguments as tuples. Every model training script is also callable from the command line that trains a model with a default hyperparameter setup. There is a small degree of interactivity as well, via command line arguments. To see available options, run the script with the `--help` argument. After finishing training, the results are logged in a file called `results.json`, in the Logs folder.

- `neural_network.py` - Trains simple neural networks
- `rnn.py` - Trains recurrent neural networks

- `other_models.py` - Trains Naive Bayes, Support vector machines and random forests.

A.1.3 utils.py

A utility script containing the definition of a Log class, that can be set up to automatically log model performance during training.

A.1.4 annotator.py

Simple script used to annotate data from Twitter and Reddit. It saves the annotations in two files ending with `_annotated.json` and `annotated_with_irrelevant.json`. The first file contains only positive and negative annotations, while the second one also contains neutral (or irrelevant) annotations. This is because at the beginning of work on the thesis, it was not decided whether to include the neutral annotations or not.

A.1.5 data_visualization.ipynb

A Jupyter Notebook used to create some visualizations out of the data.

A.1.6 dataset_processor.py

A script that combined the Twitter and Reddit datasets, while saving only some relevant data fields. The output of this script are three files:

- `train_dataset.json` - Contains annotated posts whose target sentiment is either positive or negative. There are four columns: "id" - original ID of the Reddit post/Tweet, "type" - either "reddit_post" or "tweet", denotes the origin of the post, "text" - textual content of the post, and "target" - integer representing the sentiment. Can be 0 - negative, 1 - positive, and 2 - neutral.
- `train_dataset_alt.json` - Also includes neutral posts
- `total_dataset.json` - Combined dataset of all gathered posts. The "id" field refers to the index of the post in either `reddit.json` or `twitter.json`, based on its type.

In the data folder, instead of `train_dataset.json` and `train_dataset_alt.json` we have "corrected" version of each file. This is because some annotations were incorrect, and were changed later (models have been trained on the corrected version).

A.1.7 gui.py

This script creates a GUI for model training. User first selects one of the available models, then types in hyperparameters (their descriptions can be seen via hovering over the input fields). Them the program proceeds to train the given model and save it in the Models folder.

A.1.8 enron_network.py

Creates a Networkx network out of the Enron email dataset. It iterates through all the emails, and adds the sender and the recipients (including Cc and Bcc) of the email to the network.

A.1.9 experiments.ipynb

Jupyter notebook used to run some model training experiments.

A.1.10 gamma_estimate.r

An R language script that was used for the goodness-of-fit test for both Twitter and Enron networks.

A.1.11 network_metrics.py

Calculates HITS, PageRank and Betweenness centrality for a given network, and saves in a JSON file. Also includes a function for plotting a distribution, and generating power-law distributed data. This function was not used in the end, since an implementation was already included in the poweRlaw package.

A.1.12 postprocessing.py

Resolves Twitter user IDs into names, exports networks into different file formats, and other helper functions.

A.1.13 scraper.py

Handles the gathering of posts from Reddit and Twitter. Its output are the `twitter.json` and `reddit.json` files.

A.1.14 sentiment.py

Uses a pre-trained model to estimate the sentiment of a given document corpus, in our case `total_dataset.json`. It also has an interactive mode (accessed by a command line argument `--interactive=True`). This gives the user a way to get the model prediction for console input.

A.1.15 twitter_network.py

Creates the Twitter network out of the Twitter data, and saves it.

A.1.16 pw.txt

Information for accessing Reddit API

A.1.17 twitter_info.json

Information for accessing Twitter API