Master's thesis

Review of popular word embedding models for event log anomaly detection purposes

Ville Tuulio

March 17, 2020

UNIVERSITY OF HELSINKI
Master's Programme in mathematics and statistics

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Education programme
Faculty of Science	Master's Programme in mathematics and statistics

Tekijä — Författare — Author

Ville Tuulio

Työn nimi — Arbetets titel — Title

Review of popular word embedding models for event log anomaly detection purposes

Opintosuunta — Studieinriktning — Study track

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	March 17, 2020	49 pages

Tiivistelmä — Referat — Abstract

System logs are the diagnostic window to the state of health of the server. Logs are collected to files from which system administrators can monitor the status and events in the server. The logs are usually unstructured textual messages which are difficult to go through manually, because of the ever-growing data.

Natural language processing contains different styles and techniques for a computer to interpret textual data. Word2vec and fastText are popular word embedding methods which project words to vectors of real numbers. Doc2vec is the equivalent for paragraphs and it is an extension to Word2vec. With these embedding models I will attempt to create an anomaly detector to assist the log monitoring task. For the actual anomaly detection, I will utilize Independent component analysis (ICA), Hidden Markov Model (HMM) and Long short-term memory to dig deeper in to the vectorized event log messages. The embedding models are then reviewed for their performance in this task.

The results of this study show that there is no clear difference between the success of Word2vec and fastText, but it seems that Doc2vec does not work well with the short messages the event logs contain. The anomaly detector would still need some tuning in order to work reliably in production, but it is a decent attempt to achieve useful tool for event log analysing.

Avainsanat — Nyckelord — Keywords

word2vec, doc2vec, fastText, machine learning, anomaly detection, unsupervised learning

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1. Introduction

System logs are the diagnostic window to the state of health of a server. Logs are collected to files from which system administrators can monitor the status and events in a server. The logs are usually unstructured textual messages which are difficult to go through manually, because of the ever-growing data. Often manual search is the only option for system administrators to find the cause of server faults, and there is little help from automated systems. Anomaly detectors can assist on locating the problem and in some cases, they could even predict the eventual server fault and preventive actions could be made.

Anomaly is defined as something that deviates from what is standard, normal or expected. Thus, anomaly detection from system logs is to find the rare or unexpected log message or rather sequence of log messages and this could be automated trough set of mathematical models. Anomaly detector could be used by the system administrators to detect intrusions, faulty appliance connections and unreported bugs. It could be used in cooperation with development for fixing more severe bugs which are hard to detect in normal use of the system.

"The process of event log analysis for anomaly detection involves four main steps: log collection, log parsing, feature extraction, and anomaly detection." [He+16] The raw data from the logs is unstructured, noisy and inconsistent thus some preprocessing and parsing is essential. Feature extraction is done with word embedding model which will give an event log a multidimensional vector representation. The numerical data can then be further analyzed with a set of machine learning models.

The aim in this study is to create anomaly detector by trying different vector embeddings for the feature extraction. The vector embeddings will be done with word2vec, doc2vec and fastText embedding models. Word2vec inspects each log word by word, doc2vec has broader picture comparing full log messages with each other and fastText has the ability to inspect the structure of each word of the log message individually. Further analysis and the anomaly detection are done with Independent Component Analysis, Hidden Markov Model and Long Short-Term memory. I will review the success of the embedding models for the anomaly detection task. The anomaly detector is done with collaboration of Software Point.

1.1 Motivation for this thesis

The reason for the review like thesis is, that initially I wanted to create event log anomaly detector which could be used generally for different applications. Doc2vec embedding model seemed powerful tool to transform unstructured log messages to vector embeddings which could be then further processed with other machine learning models. However, I noticed that the log vectors created by Doc2vec didn't converge towards similar log messages and seemed that each vector is just a randomly distributed vector. Even with longer training times the convergence was quite low and only log messages with substantially more words than others could be distinguished from each other. Since Doc2vec is not the only embedding model available I wanted to try with some other models too and writing a review of them seemed to be an interesting pursuit.

1.2 Structure of the thesis

First in this thesis the word embedding models are introduced. Next the proposed anomaly detection models are represented with mathematical explanations. Then I display how these models are trained and used in this thesis in order to solve the anomaly detection problem. After this the results of the embedding and the anomaly detection models are represented and reviewed. Lastly open discussion about the results and future work suggestions.

2. Word embedding

Natural language processing (NLP) is an interdisciplinary field, which combines linguistics and artificial intelligence in pursue of making computers read and understand human language in a manner that valuable artefacts could be produced. For example, speech recognition and text classification are of NLP's area of interest. Text classification tasks could be dividing movie reviews to positive and negative[CZ05] and even detecting influenza epidemics by mining twitter messages [Cul10]. The problem at stake in this thesis is to understand whether a textual event log message is anomalous.

Since the anomaly detection algorithms need structured vector like inputs, some kind of feature extraction is needed on the log messages. One of the simplest word vectorization is one-hot encoding. One-hot encoding creates a vector for each word in the vocabulary, where all of the values in the vector are marked as zero expect for the index of the word is marked as one. However, one-hot encoding explains nothing about the semantics of the words and each vectorization is just an orthogonal representation to another dimension. Since each new word creates new dimension for the vector the computational toll for anomaly detection would be severe. Thus, word embedding models could bring faster solution for this.

Word embeddings contains a class of techniques where each word is represented by a real-valued vector, often tens or hundreds of dimensions. This is contrasted to the thousands of dimensions required for a one-hot encoding. The values of the vector are acquired by learning a supervised machine learning model such as Word2vec.

The word embedding models used in this study are presented in the next subsections.

2.1 Word2Vec

Word2vec is a vector embedding model which is an efficient method for learning high quality vector representations of words from large amounts of unstructured text data. Since the training of the model does not involve dense matrix multiplications, the training is rather efficient. "An optimized single-machine implemen-

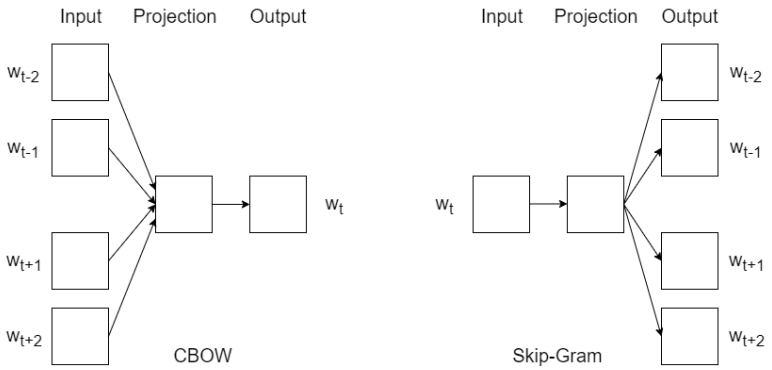

Figure 2.1: The CBOW and Skip-gram model frameworks. The training objectives are to learn word vector representations that are good at predicting the middle word (CBOW) or the nearby words (Skip-gram). [SAA17]

tation can train on more than 100 billion words in one day." [Mik+13] Word2Vec is a two-layer neural network. It takes words as one-hot vectors, the vector is feedforwarded trough fully-connected layer's weights and it outputs probabilities of target words from the vocabulary of the corpus. Word2vec has two different architectures of finding the word vectors: Skip-gram and Continous Bag-of-Words (CBOW). The difference is that CBOW tries to predict the middle word given surrounding words and Skip-gram tries to predict surrounding words given the middle word. In this study the Skip-gram model with negative sampling is used and it is described as follows:

To train the model the objective is to maximize the log probability of

$$\sum_{t=1}^{T} \sum_{c \in \mathcal{C}_t} \log p(w_c|w_t), \tag{2.1}$$

where $\mathcal{C}_t$ is the training context (the words surrounding center word w_t). Larger $\mathcal{C}_t$ results in more training examples and thus can lead to a higher accuracy, at the expense of the training time. However, the size of $\mathcal{C}_t$ can be in the range $5 - 20$ for small training datasets and for larger datasets the size of $\mathcal{C}_t$ can be smaller [Mik+13]. The problem of predicting context words can be set of independent binary classification tasks. Then the goal is to independently predict the presence of context words. Using binary logistic loss, the negative log-likelihood $-\log p(w_c|w_t)$ can then be defined by as follows:

$$\log(1 + e^{-s(w_t,w_c)}) + \sum_{n \in \mathcal{N}_{t,c}} \log(1 + e^{s(w_t,n)}). \tag{2.2}$$

Here the $N_{t,c}$ is a set of negative examples sampled from the vocabulary and the

s is a scoring function which maps pairs of word and context to scores in $\mathbb{R}$. Now we can write the objective as follows:

$$\sum_{t=1}^{\mathcal{T}}[\sum_{c\in\mathcal{C}_t}\log(1+e^{-s(w_t,w_c)})+\sum_{n\in\mathcal{N}_{t,c}}\log(1+e^{s(w_t,n)})] \tag{2.3}$$

Parametrization for the scoring function s between the center word w_t and the context word w_c is to use word vectors. For each word w in the vocabulary there are defined two-word vectors: u_w and v_w. These are the "input" and "output" vector representations of w. In particular the vectors u_{w_t} and v_{w_c} corresponding to words w_t and w_c. Then the score can be computed as scalar product of $s(w_t, w_c) = u_{w_t}^\top v_{w_c}$.

Word2vec has interesting feature, a word offset technique where simple algebraic operations are performed on the word vectors, it was shown for example that vector("King") - vector("Man") + vector("Woman") results in a vector that is closest to the vector representation of the word "Queen" [MYZ13].

2.2 fastText

Word2vec treats each word as unique and ignores the internal structure of words. [Boj+16]. Event log messages might contain file names, rare or unique words and even misspelling. Each unique word will be handled as a new token in the vocabulary even though the meaning of the word could be same as with another word. fastText is possible solution for this and it is an extension to the Word2vec model. Instead of learing vectors for each individual word, fastText represents each word as an n-gram of characters. Boundary symbols < and > will be added at the beginning and end of a word, allowing it to be distinguished from other words. Also, the word itself is included in the set of n-grams. Taking a look at the word *update* and setting $n = 4$ as an example, the n-gram will then be:

$$< upd, upda, pdat, date, ate > .$$

Now supposedly a dictionary of n-grams of size G is given. With word w the set of n-grams is $\mathcal{G}_w \subset \{1, \ldots, G\}$. A vector representation $\mathbf{z}_g$ is presented for each n-gram g. The overall word embedding is the sum of these character n-grams thus the scoring function is:

$$s(w, c) = \sum_{g\in\mathcal{G}_w} \mathbf{z}_g^\top \mathbf{v}_c \tag{2.4}$$

Since fastText breaks words down to n-grams it can find good word embeddings to rare words and even words that were not seen during training.

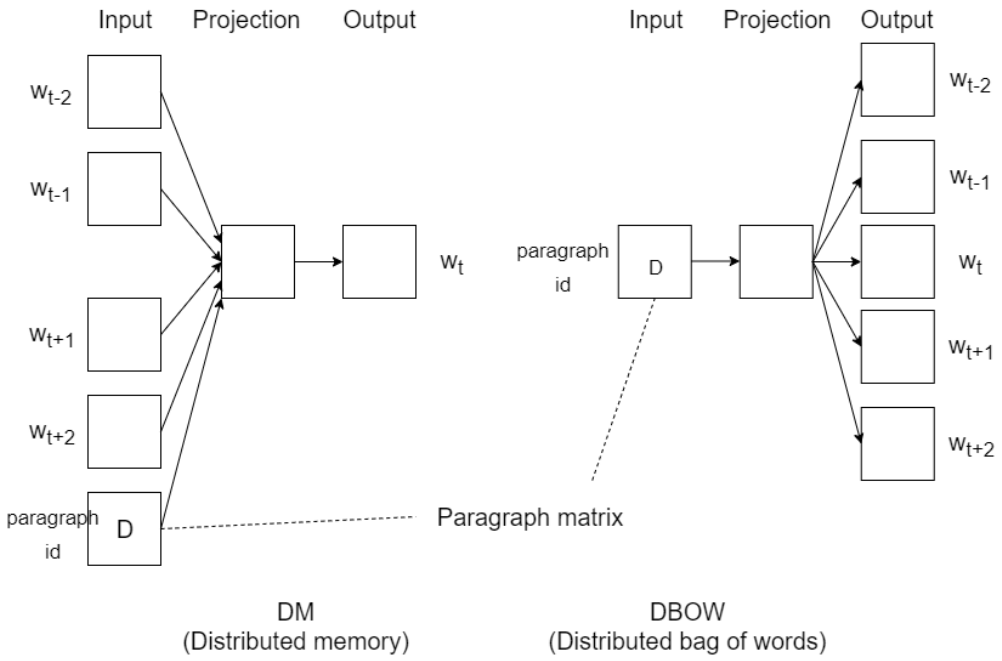

Figure 2.2: Distributed Memory and Distributed Bag of Words frameworks for doc2vec. The Distributed Memory model is similar to the word2vec CBOW model. Distributed Bag of Words is similar to the word2vec Skip-gram model.

2.3 Doc2Vec

Doc2Vec, also known as *Paragraph Vector*, is similar algorithm to Word2vec and fastText, but instead of vectorizing words it creates vector embedding of pieces of text, such as sentences, paragraphs, and documents. Doc2vec is also an extension of Word2vec and it was introduced by Le and Mikolov [LM14]. The doc2vec model also has two different variations Distributed Memory and Distributed Bag of Words. Distributed Memory model is similar to the Word2vec CBOW mode where the only change is an additional paragraph token that is mapped to a vector. The paragraph token can be thought as another word vector in the input which represents missing information from the current context.

In Distributed Bag of Words version, the model tries to predict word vectors randomly sampled from the current paragraph. This way each iteration a text window is sampled from which a random word is sampled, and a classification task is formed given the paragraph vector.

Doc2vec has shown some promising results in event logs anomaly detection models which are based on natural language processing. [Mim19]

2.4 Random vector embedding

In addition to the previous embedding models a random vector embedding is added. Random vector embedding creates a random vector for each word in the vocabulary from which the log vectors can be composed. Random vector embedding is essentially Word2vec with no training. This makes each log vector different in a way that messages such as:

```
2019-03-18 10:09:36,459 [127.0.0.1] INFO  ELEMENTTAG  4340 - Generating HTML
2019-03-18 10:09:36,479 [127.0.0.1] INFO  ELEMENTTAG  4340 - Done generating HTML
```

might be embedded completely differently even though the semantics are similar. Random vector embedding is done mainly for base line measurement. If random vector embedding succeeds better than any other embedding method, then there is no reason to use time and processing resources to embed log messages for anomaly detection.

3. Anomaly detection

Anomaly detection in event log analysis leans heavily on pattern recognition. A "normal" pattern is attempted to find and if an event doesn't follow the pattern it may be considered as an anomaly. In order to create useful detector, the threshold of anomaly must be set carefully. It is not intentional to overwhelm the system administrator with hordes of false positives.

In this study there are three different types of anomaly detection models. First one is the only fully unsupervised model called Independent Component Analysis (ICA). ICA attempts to separate the data to statistically independent sources. The preprocessing of ICA is done by centering and whitening the data. During whitening, the higher dimensional log vectors are projected to lower dimensions to help the clustering problem. The datapoints which don't fit well in any cluster are considered outliers and may be anomalous log messages. An outlier is not necessarily anomalous since it could be just a rare event. Thus, this model is more of supportive type for the other models. Perhaps the best known problem solved with ICA is the "cocktail party problem", where the goal is to separate independent signal from noisy mixture of signals. However ICA has also shown promises in server log anomaly detection [LZL10].

Usually in event logs the interesting objects are not just rare events but rather the unexpected sequence of events. Thus, the second model is Hidden Markov Model (HMM). HMM assumes the system to be a Markov Process with unobservable states. This means that the output of the system can be observed but it is considered to be a result of the hidden unobservable state. Each state has some probability distribution for the emitted output and a probability to transition to different state. The model could be initialized without training, but one would then need to know the emission and transition probabilities. Since the transition and emission matrices are unknown a Baum-Welch training algorithm is used to initialize the matrices. HMM has been used largely for speech recognition, but it has also succeeded in system anomaly detection [WFP99; YM05]

Lastly a recurrent neural network model, Long short-term memory (LSTM), is used to inspect the log vectors. LSTM is used as sparse autoencoder where the model maps an input sequence to a fixed-length vector, and then maps it again

to the same size as the input. Anomalies are such log messages or sequences of messages which the auto-encoder finds difficult to predict. Similar auto-encoder has already been successfully used in text classification [Xu+17], text generation [LLJ15] and time-series anomaly detection [Mal+16].

In the next subsections all the three models are introduced in more depth.

3.1 Independent Component Analysis

Independent Component Analysis is a statistical model which transforms the given multidimensional data into components which are statistically as independent as possible. ICA was chosen over Principal component analysis (PCA) since it has been shown that ICA can more effectively identify anomalies in some cases [LZL10]. ICA is also a special case of blind source separation and it might be of use for the other anomaly detection models.

The ICA is a generative model and it tries to explain how the observed data was generated by mixing process of some independent components. Thus ICA assumes that the embedded log vectors $x = (x_1, \ldots, x_n)$ can be explained by independent components $s = (s_1, \ldots, s_m)$ with a linear transformation matrix which is called a mixing matrix $\mathbf{A}$ in this context. With the variables we get the statistical model of ICA as,

$$x = \mathbf{A}s. \tag{3.1}$$

To solve the independent components, an unmixing matrix $\mathbf{W}$, such that $\mathbf{WA} = \boldsymbol{I}$, need to be found. Then the independent components are obtained by,

$$s = \mathbf{W}x. \tag{3.2}$$

Many of existing algorithms for ICA aim to maximize the non-gaussianity of the independent components. One way to find the mixing matrix $\mathbf{W}$ is with the Algorithm 1. The algorithm is called FastICA, which was introduced by Aapo Hyvärinen [Hyv99; HO00] and it is one of the most widely used algorithms performing ICA.

To simplify the algorithm, the data should be preprocessed first by centering and second by whitening it. Centering can simply be done by subtracting each element by the mean vector $m = E(x)$. The data is whitened by a linear transformation so that the new vector's $\hat{x}$ components are uncorrelated and their variances equal unity. Thus the covariance matrix of the data equals the identity matrix $E(\hat{x}\hat{x}^\top) = \mathbb{I}$. Whitening can be done with eigen-value decomposition [HO00].

Algorithm 1: FastICA

Input: The data x, number of independent components k
Result: Unmixing matrix $\mathbf{W}$
Center and whiten the data x
Initialize the weight vectors $\mathbf{W} = (w_1, \ldots, w_k)$ randomly
repeat
  for $j = 1$ *to* k **do**
    Update $w_j^+ = E\{xg'(w_j^\top x)\} - E\{g''(w_j^\top x)\}$ for $\forall j \in \{1, \ldots, k\}$
  end
  Update $\mathbf{W} = w^+/|w^+|$
until *convergence*;

The function g is non-quatratic function to measure the non-Gaussianity. For example function $g_1(u) = \frac{1}{a_1} \log \cosh(a_1 u)$ with values $1 \leq a_1 \leq 2$ is good for general purposes contrast function, and function $g_2(u) - \frac{1}{a_2} e^{-a_2 u^2/2}$ for more robust solutions [HO00]. Convergence is achieved when the dot-product of old and new values of w is close to 1.

For log anomaly detection purposes, the ICA can separate anomalous sources from the data and the preprocessing reduces the dimensionality of the log vectors for making the clustering easier. An outlier could be considered as anomaly although, the outlier detection is only good for so far since many interesting log events are inside a sequence of events and not the individual logs itself. In the next section the Hidden Markov Model is introduced which offers a solution to the sequential data.

3.2 Hidden Markov Model

A Markov Chain or Markov Model is a stochastic process where a sequence of changing states is observed, and it is assumed that the future state is independent of past state given the present state. In Hidden Markov Model (HMM) the changing states cannot be observed but each state emissions some observable output, from which it can be predicted in which state the model is. Since hidden states changes and if the transition probability is known the next state can be predicted. A simple example of HMM is a doorman who cannot see the weather outside but can see what people are wearing when they come in. The hidden state here is the weather and the observations are people's attire. If many people are observed to carry an umbrella and wearing raincoats the doorman can predict that the hidden state alias the weather is rainy. Knowing the current state, doorman can predict that the next day will likely be rainy or cloudy and it is unlikely to be sunny.

In attempt to transit this idea for log anomaly detection a set of log vectors are observed which are usually seen in unhealthy system. Then a prediction can be made that the system is unhealthy or in transition to unhealthy state. Even if it is unknown what is the difference of healthy or unhealthy log vectors, the probability of observations can be calculated, and log vectors can be assumed as anomalies based on that.

A HMM is specified by a number of states **N**, their initial probability distribution π, a transition probability matrix **A** and an emission probability matrix **B**. The transition probability matrix tells the probability of moving from state i to state j. The emission probability matrix tells the probability of an observation to be emitted from a state. The model is noted as $\lambda = (\mathbf{A}, \mathbf{B}, \pi)$ from now on. HMM assumes that the sequence of states is a Markov Chain and that the probability of each observation depends only on the state that produced the observation. The paper "Introduction to Hidden Markov Models" [RJ86] gives HMM three key problems of interest that must be solved for the model to be useful in real worlds applications. These problems are the following:

Problem 1: Given the observation sequence $O = (O_1, \ldots, O_T)$ and the model λ, how do we compute the probability of the observation sequence.

Problem 2: Given the observation sequence $O = (O_1, \ldots, O_T)$, how we choose a state sequence $I = (i_1, \ldots, i_T)$ which is optimal in some meaningful sense.

Problem 3: How can we set the model parameters $\lambda = (\mathbf{A}, \mathbf{B}, \pi)$ to maximize the probability of observation sequence $O = (O_1, \ldots, O_T)$ given λ.

In short, the problems ask, how to calculate the probability of a sequence, how to predict the current hidden state and how to initialize and train the model? The solutions for the problems are stated next.

3.2.1 Forward algorithm

To calculate the probability of observed sequence O with the model λ, the direct calculation would be by adding the emission probabilities of every possible state sequence. For an HMM with N hidden states and sequence of length T there are N^T possible state sequences. So, the direct calculation will be unfeasible even for small values of N and T. Instead of direct calculation, a dynamic programming algorithm, the Forward algorithm, is used instead. For the algorithm a forward variable $\alpha_t(i)$ needs to be defined as follows:

$$\alpha_t(i) = P(O_1, \ldots, O_t, i_t = q_i | \lambda) \tag{3.3}$$

This is the probability of observation sequence from observation O_1 to observation O_t and state q at time t, given the model λ. The variable $\alpha_t(i)$ can then be solved with the Algorithm 2.

Algorithm 2: Forward algorithm

Input: The observed sequence O, the HMM model $\lambda = (A, B, \pi)$
Result: Total probability of the sequence given the model:
$P(O|\lambda) = \sum_{j=1}^{N} \alpha_T(i)$
Calculate the first forward variable
$\alpha_1(i) = \pi_i b_i(O_1)$, for $1 \leq i \leq N$
The rest variables can be calculated as
for $t = 1, \ldots, T-1$ **do**
  for $1 \leq j \leq N$ **do**
    $\alpha_{t+1}(j) = \sum_{i=1}^{N} \alpha_t(i) a_{ij} b_j(O_{t+1})$
  end
end

Here α_{ij} is the transition probability from state i to state j and $b_j(O_t)$ is the state observation probability of the observation O_t given current state j. Each probability α_t contains the sum of probabilities $(\alpha_{t-1}, \alpha_{t-2}, \ldots, \alpha_1)$ and to calculate the next probability α_{t+1} one only needs to reuse the current α_t and add the transition and emission probability for every state.

3.2.2 Viterbi algorithm

For HMM the hidden states are usually the interesting part. In particular which state the model is after some observations are observed? Again, by direct calculation of each possible hidden state sequence and maximizing the likelihood this is doable, but it is frustratingly tedious since the sequences for hidden states increases exponentially with the number of hidden states. Also, here a dynamical programming algorithm, the Viterbi algorithm, is proposed. Viterbi algorithm is similar as the Forward algorithm but instead of calculating the sum of probabilities Viterbi takes the maximum of the previous state probabilities. Also, Viterbi doesn't only return the probability of the sequence, but it also returns the sequence itself. The Viterbi algorithm is defined in Algorithm 3.

Further explanation of the Viterbi algorithm can be found from the paper [For73].

Algorithm 3: Viterbi algorithm

Input: The observed sequence O, the HMM model $\lambda = (A, B, \pi)$

Result: The probability of the most probable hidden state sequence and the sequence of hidden states.

Initialize the variables δ and Ψ:

for $1 \leq i \leq N$ **do**

$\delta_1(i) = \pi_i b_i(o_1)$

$\Psi_1(i) = 0$

end

Recursive calculation:

for $2 \leq t \leq T$ **do**

for $1 < j \leq N$ **do**

$\delta_t(j) = \max_{1 \leq i \leq N} \delta_{t-1}(i) a_{ij} b_j(o_t)$

$\Psi_t(j) = \arg\max_{1 \leq i \leq N} \delta_{t-1}(i) a_{ij}$

end

end

Termination:

$P^{max} = \max_{1 \leq i \leq N} \delta_T(i)$

$i_T^{max} = \arg\max_{1 \leq i \leq N} \delta_T(i)$

Sequence of the most probable states:

for $t = T-1, T-2, \ldots, 1$ **do**

$i_t^{max} = \Psi_{t+1}(i_{t+1})$

end

3.2.3 Baum-Velch algorithm

In order to get meaningful results with the HMM the model parameters need to be initialized with reasonable values. However, the emission or transition distribution is often unknown, as in the case of log anomaly detection. Thus, a solution for setting and training the parameters needs to be found. One of the most used HMM parameter estimator is the Baum-Velch algorithm. It is a special case of the expectation-maximization algorithm (EM-algorithm) and its purpose is to set the transition matrix $\mathbf{A}$, emission matrix $\mathbf{B}$ and the initial state distribution π such that the probability of observations $P(O|\lambda)$ is maximized. Baum-Welch algorithm utilizes a two-part algorithm, the Forward-Backward algorithm, from which the first part, Forward-algorithm was introduced in previous subsection. Thus, next is the Backward part.

Backward-algorithm: Backward algorithm is quite similar to the Forward algorithm but it deals with the observations in reversed order. The backward variable can be defined as:

$$\beta_t(i) = P(O_{t+1}, O_{t+2}, \ldots, O_T | i_t = q_i, \lambda) \tag{3.4}$$

This is the partial observation sequence from $t+1$ to the end of the full sequence. Given are the state q_i at time t and the model λ. The variable $\beta_t(i)$ can be solved with the Algorithm 4.

Algorithm 4: Backward algorithm

Input: The observed sequence O, the HMM model $\lambda = (A, B, \pi)$
Result: Total probability of the sequence given the model:
$P(O|\lambda) = \sum_{j=1}^{N} \pi_j b_j(o_1)\beta_1(i)$
Initialize the end of the observation sequence:
for $1 \leq i \leq N$ **do**
  $\beta_T(i) = 1$
end
Probability of each observartion in the sequence:
for $1 \leq i \leq N$ **do**
  for $1 \leq t \leq T$ **do**
    $\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(O_{t+1})\beta_{t+1}(j)$
  end
end

The first step defines the β_T arbitrary to be 1 for all i and the second step is the recursive step which notes that the result of $\beta_t(j)$ is equal to sum of all

successive values $\beta_{t+1}(j)$ multiplied by transition and observation probabilities, a_{ij} and $b_j(O_{t+1})$.

Now that both α_t and β_t are solved the Baum-Welch with the Forward-Backward algorithm can be defined as the Algorithm 5.

Like the EM-algorithm the Baum-Velch has two phases in which the algorithm will alternate. In the E-step the expected state occupancy count γ and the state transition count ξ are calculated with the transition matrix $\mathbf{A}$ and the emission matrix $\mathbf{B}$. In the M-step the values γ and ξ are used to recompute new values for matrices $\mathbf{A}$ and $\mathbf{B}$. The matrices $\mathbf{A}$ and $\mathbf{B}$ and the distribution π need to be initialized randomly if there is no further information about the observation sequences.

The HMM gives unsupervised approach for finding anomalies from the sequential data. First the model is initialized by feeding the data to Baum-Welch algorithm and then analyzed with the Forward and Viterbi algorithms. Forward algorithm gives the probability of given sequence and it is used to compare adjacent or parallel sequences probabilities. If one's probability is a lot lower than the others, then the sequence can be assumed anomalous. Viterbi algorithm is used to find the hidden state sequence and if one state can be thought anomalous then logs which are emitted from the anomalous state are assumed anomalous.

A competing and more complex method for sequential anomaly detection is the Long Short-Term Memory and it is hypothized to perform better than the HMM. Although it is been shown that despite the success of deep neural networks, a more traditional graphical model approaches can be beneficial [PC16; MHA00].

Algorithm 5: Baum-Welch algorithm

Input: The observations O_T, hidden state set Q_N

Result: Trained HMM model $\lambda = (A, B, \pi)$

Initialize A and B randomly

repeat

  for $1 \leq t \leq T$ **do**

    for $1 \leq j \leq N$ **do**

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{P(O|\lambda)}$$

      for $1 \leq i \leq N$ **do**

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{P(O|\lambda)}$$

      end

    end

  end

  for $1 \leq i \leq N$ **do**

    estimate the initial states:

$$\bar{\pi}_i = \gamma_1(i)$$

    estimate the transition matrix:

$$\bar{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

    estimate the emission matrix:

$$\bar{b}_j(k) = \frac{\sum_{t=1,O_t=k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

  end

until *Until* $P(O|\lambda) \approx P(O|\hat{\lambda})$;

3.3 Long Short-Term Memory

Recurrent neural networks (RNN) are used to learn patterns from sequential data and make predictions out of it. RNN can memorize parts of the input during the training sort of creating hidden state for the input and use them later to make predictions. Nonetheless during RNN's training it has hard time to remember inputs that are far away from the sequence. For example, if RNN is trying to process a long text to do some prediction it might leave important bit from the beginning and tend to make predictions only on the last words. This problem is also called the vanishing gradient problem. Long Short-Term Memory (LSTM) architecture holds solution for this since it has memory cell that can store information from longer sequences. It is the state-of-the-art for variety of machine learning problems and especially learning problems related to sequential data [Gre+17]. Since LSTM is a special case of RNN a short description to RNN is in place.

3.3.1 Recurrent Neural Network

A Traditional neural network assumes that all inputs are independent of each other. In many cases that is not the case. For example, in weather prediction a cold day is unlikely to be followed by a warm day, instead it would be mild or something between. RNNs passes a hidden value from previous layer to the next, thus all inputs are assumed to be dependent of each other. Prediction with RNN is calculated so that, each input x_t from the sequence $X = (x_1, \ldots, x_T)$ is combined with a hidden vector h_t from the sequence $h = (h_1, \ldots, h_T)$ by equation:

$$h_t = \sigma(Wx_t + Uh_{t-1} + b_h). \tag{3.5}$$

Here σ is the activation function, W are the input weights, U are the hidden weights and b is the bias vector. Output can be calculated then with:

$$y_t = \sigma(Vh_t + b_y). \tag{3.6}$$

Here the matrix V denotes the output weights. Depending on the prediction task the output can be calculated for every input value or smaller subset of input values. For example, a machine translation task would calculate the output for each input and a sequence classification task would calculate the output for just the last input. When training RNN the loss of each output value needs to be taken into consideration. Thus, the loss for one output sequence is the sum of each output's loss:

$$L = \sum_{i=1}^{T} d(y_i, \hat{y_i}) \tag{3.7}$$

Since each input is dependent on each other the backpropagation needs to be done at each point in time. At time-step T the gradient of loss L is calculated as:

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}. \tag{3.8}$$

Here the right-hand term is by the chain rule of derivation as follows:

$$\frac{\partial L_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}. \tag{3.9}$$

Finally the weight update is done as with the update rule $\hat{W} = W - \alpha \frac{\partial L}{\partial W}$. Here α is the learning rate which is set manually. This backpropagation trough every time-step is also known as backpropagation trough time (BPTT). The problem with vanishing gradients comes up in the training since the term $\frac{\partial h_t}{\partial h_k}$ is factorized as follows:

$$\frac{\partial h_t}{\partial h_k} = \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_{k+1}}{\partial h_k} = \prod_{j=k}^{t} \frac{\partial h_j}{\partial h_{j-1}} \tag{3.10}$$

When $t-k$ increases, the size of the factorized sequence converges quickly to 0, since the term $|\frac{\partial h_t}{\partial h_k}| < 1$. Thus the gradient, and the weight updates, are dominated only by the inputs closer to the last input of the training sequence. Vanishing gradient might prolong the training extremely and it might not converge at all. If the term $|\frac{\partial h_t}{\partial h_k}| > 1$ then the training would be wildly unstable for gradients are increasing rapidly. This issue is also known as exploding gradients.

As stated before, LSTM brings answer to the vanishing and exploding gradient problem and it is handled in the next section.

3.3.2 Long short-term memory

LSTM was introduced by Horchreiter and Schmidhuber in 1997 [HS97] and it is a RNN architecture that contains units called memory blocks in the recurrent hidden layer. The memory blocks are capable to store the temporal state and control the flow of information through functions which are also called as gates. The memory blocks contain input, forget, output gates and traditional activation functions. Gates composed out if a sigmoid neural net layer and a point-wise multiplication operation. Gates either let information go through or block the information flow and it basically regulates how the cell state is changed during the process. The LSTM network iterates trough $1 \leq t \leq T$ mapping the input sequence $x = (x_1, \ldots, x_T)$ to an output sequence $y = (y_1, \ldots, y_T)$ using the following functions:

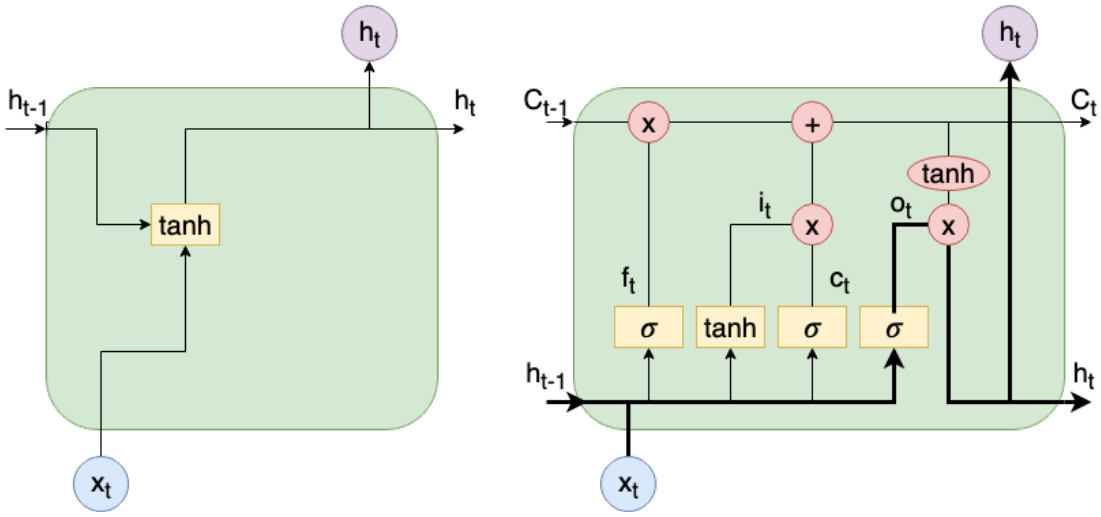

Figure 3.1: Comparison between RNN and LSTM architecture.

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \tag{3.11}$$
$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \tag{3.12}$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \tag{3.13}$$
$$a_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \tag{3.14}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot a_t \tag{3.15}$$
$$h_t = o_t \odot \sigma_h(c_t) \tag{3.16}$$
$$y_t = \phi(V h_t + b_y) \tag{3.17}$$

Here i_t, f_t and o_t are the input, forget and output gates, a_t is the input activation function vector, c_t stands for the cell state and h_t is the cell output or the hidden state. σ_g is sigmoid function, which is used in all of the gates, σ_h and σ_c are hyperbolic tangent activation functions. $W = [W_i, W_f, W_o, W_c]^\top$ and $U = [U_i, U_f, U_o, U_c]^\top$ is the concatenated weight matrix for input and output weights. Matrix V is the output weights. σ_g and σ_h are the cell input and output activation functions, ϕ is the softmax activation function and $\odot$ is element-wise product of the vectors. Initial values of c_0 and h_0 are 0.

In order to train the network, the process is similar as with RNN. The loss is calculated as in equation 3.8 and 3.9 but now the gradient is a bit different. At

time step t the derivation of the cell state is calculated as:

$$\frac{\partial c_t}{\partial c_{t-1}} = \frac{\partial}{\partial c_{t-1}}[f_t \odot c_{t-1} + i_t \odot a_t] \tag{3.18}$$

$$= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \tag{3.19}$$
$$+ \frac{\partial}{\partial c_{t-1}} \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \odot \sigma_g(W_i x_t + U_i h_{t-1} + b_i).$$

Here the forget gate dominates the result of the gradient and if the gate decides that certain piece of information should be remembered then $f_t \approx 1$. Thus the cell state gradient is $\frac{\partial c_t}{\partial c_k} \nrightarrow 0$ and the gradient of the loss function $\frac{\partial L_t}{\partial W} \nrightarrow 0$. This proofs that the gradients do not vanish. Again the weights are updated by the update rule: $\hat{W} = W - \alpha \frac{\partial L}{\partial W}$.

Even though LSTM is huge improvement to the regular RNN it still might struggle with very long-time dependencies. LSTM also scales quickly making training time consuming even with faster computers.

4. Methodology

In this chapter it is presented how the embedding and anomaly detection models are trained and combined for the anomaly detection attempt. The structure of this chapter will follow the four main steps of event log anomaly detection [He+16], log collection, log parsing, feature extraction and anomaly detection.

Event logs are collected automatically from the servers, and a set of 484280 log messages are used for training the embedding models. These messages are then further analysed with the anomaly detection models. Prior knowledge of the logs is that during the logging time the server had lost connection from the database twice. This is the interesting part which would be nice to be found with the anomaly detection models, but other interesting events might also be found from the logs. Before the logs are inputted to the embedding models, some preprocessing is to be made.

4.1 Data preprocessing

Usually event log message contains at least the following attributes: timestamp, thread, severity, action name, and free-form message text. For the study and the embedding models, interesting attributes are all but the timestamp. Thus, the textual inputs for the embedding models will be of type:

```
[THREAD] [SEVERITY] [ACTION] [MESSAGE]
```

The thread, severity and action name are fixed values and they do not need further handling. Preprocessing is needed for the free-form message part, even though the embedding models could probably learn the training data without any preprocessing. The data contains messages that are similar to each other but there could be a small detail which distinguishes the message from another. One example is messages that contains a duration of action call.

```
[thread-1] INFO  SecurityCheck - Statement: Department Security check for 'TrackItem' having '1'

rows with '1' action calls took 1.93ms
```

```
[thread-1] INFO  SecurityCheck - Statement: Department Security check for 'TrackItem' having '1'
rows with '1' action calls took 1.87ms
```

These messages deliver the same message with only slight time difference, nonetheless the messages are currently considered different according to the embedding models. So it is proposed that the messages will be modified to help unify certain events. In this case the preprocessed message is changed to:

```
[thread-1] INFO SecurityCheck statement department security check for trackitem having <int>
rows with <int> action calls took <float> ms
```

Note that the message is also converted to lower case and apostrophes and colons are removed. Also, only the message part of the log is changed and other parts are unmodified. The table 4.1 contains all message conversions and short reasoning why the conversion is made. It is debatable whether preprocessing is needed, since the embedding models would probably give these logs messages vector embeddings that are close for each other even without any preprocessing. However, the preprocessing will make the embedding process a substantially easier when there are less words in the vocabulary.

Conversion	Reason
All text to lower case.	The text is free written text some of the word tokens might have capital letters and thus handled as unique word.
Symbols such as " ' : ; . , replaced with space, and extra whitespaces are removed.	Some words end in dot or colon or they are surrounded by parenthesizes. Removing them makes vocabulary smaller and simpler.
Dates to "[date]" or "[now]" if the date is really close to the log timestamp	Some messages contains dates. Now is set if the date in the message is close to its timestamp.
Highlights such as "+++++++" to "[highlight]".	There were different sized highlighters and with different symbol sequences. However, the use of it is that it is easier to spot for the administrator and the lengths or different symbols doesn't matter.
Connection id to "[id]".	Some messages include the connection id of the user. It's better to simplify this by replacing the changing sequence with id token.
Numbers are changed to "[serie]", "[float]" or "[int]" depending of the number	Messages include series (1.2.4.33.1) floats (1.87) and integers (60) they all are changed since the log file might contain too much unique words without conversion.

Table 4.1: Table of log message conversions used.

4.2 Feature extraction

Each embedding model, Word2vec, fastText, Doc2vec and random vector embedding, are trained to produce 100- and 300-dimensional vectors and each model will be trained in total of 20 epochs. The attempt with chosen dimensions is to find the boundary of how many is needed in order to explain the data reasonably. Each trained model has sliding window size of 5 words. In Word2vec and fastText there is no break between new log message, instead their training data is concatenated from all of the log messages creating a long document like structure. Since Word2vec, fastText and random vector embedding creates word vectors instead of vectors from full sentences, like Doc2vec, the word vectors from a log message need to be combined to create a log vector. Log vectors are created by simply summing all word vectors from a log with each other. For the Doc2vec model each log message is considered as new document in its training corpus.

The data used for training has a lot of duplicate log messages and during the implementation phase it was discovered that Doc2vec model could not distinguish same or even similar messages from each other. This is probably due to the nondeterministic nature of the model. Each log vector is randomly initialized. Also, Doc2vec is better suited for finding embeddings to longer documents instead of typical log message with only than 10 words or less. Even though the log vectors from duplicate messages should be converging towards each other and it was tested by increasing the training epochs, but it didn't have significant effect. The figure 4.1 shows comparison between Word2vec and Doc2vec embedding models which illustrates the problem. An alternative approach for using Doc2vec is introduced in order to find better embeddings with the model.

A vocabulary is created from each unique preprocessed log message and a Doc2vec model is trained against this vocabulary. This shall be called *unique Doc2vec* model in this thesis. The *unique Doc2vec* model is also trained with 20 epochs but since it is trained against each unique log message the training data is only 12733 messages instead of the 484280 messages. Adding the *unique Doc2vec* models to the review pool, there will be in total of 10 different embedding models to compare.

4.3 Anomaly detection

Each of the anomaly detection models are unsupervised. The unsupervised approach is useful since it doesn't need pre-labeled data. With unsupervised models it is also easier to generalize the models to work for event log data from different applications. As downside unsupervised learning has the cost of usually needing more training data, hence more training time.

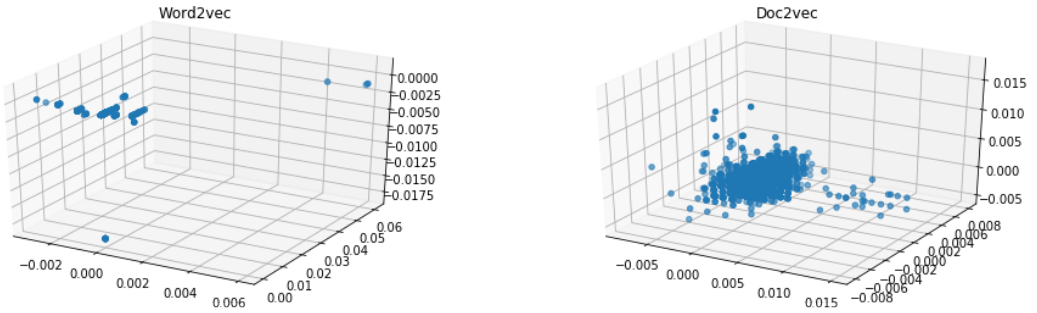

Figure 4.1: Comparison between Word2vec and Doc2vec. The nondeterministic nature of the Doc2vec and short sentences makes it difficult to achieve dense representation from each unique log message.

It should be noted that the 484280 logs contain 12733 unique messages from where many messages are similar with only few word differences. Thus, the data contains limited amount of features, which means that the amount of clusters and hidden states for the ICA and HMM anomaly detection models can probably be set fairly low compared to the size of the data.

4.3.1 Independent Component Analysis

The ICA is used for separating the independent data components to possibly detect easily anomalous signals. Also the preprocessing for ICA projects the 100- and 300-dimensional data to 3 dimensions which is then further analyzed with a clustering algorithm. The data is clustered to 100 clusters in total, with k-means clustering algorithm. If an event log does not fit well in any of the clusters, it is labeled as unclustered. The unwell fitting is determined by the log vectors distance from the center of cluster and the limit distance is the maximum distance between any of two centers of clusters.

These clusters are further analyzed with the HMM model and the lower dimensional vector representation is used in LSTM.

4.3.2 Hidden Markov Model

Instead of log vectors the HMM uses the clusters from ICA model as training data. Hence the sequences given to the HMM are sequences of cluster indexes. The model is initialized with 10 hidden states and the emission and the transition matrices are initialized with uniform random probabilities. The model parameters are then trained with the Baum-Welch algorithm. The training data is the complete available data of 484280 cluster indexes which is split to shorter sequences for each unique connection id.

There are two ways to determine anomalous logs with HMM and both ways use the Viterbi algorithm. We input the Viterbi with a short sequence of logs where the examined log is in the middle. The whole data can be examined with sliding window technique so that for each log a sequence is created with total of 20 and 100 logs. This way the probability of the sequences and the hidden state of the log can be calculated. Probabilities of the sequences will be compared with each other and if one is more unlikely than the other then the sequence hence the log might be anomalous.

The other way to determine anomalous log is to inspect the hidden states given by the Viterbi algorithm. Even though it is initially unknown which states are anomalous we could determine anomalous state by low transition probabilities or if the state's emission probabilities separate a lot from the other states.

4.3.3 Auto-Encoder (LSTM)

The LSTM is used as sparse autoencoder. Sparsity helps the Auto-Encoder model to avoid overfitting and it regularizes better. Before the detailed explanation of the model here is a short motivation to use dropout to achieve sparsity for the model.

Dropout: When the hidden layers have larger size than the input vector and it has the possibility to learn the identity function, making the anomaly detector useless. Sparsely connected layers could avoid this, but the more robust way is to use dropout. This means that only some hidden units in a layer are active during the forward/backward pass and the majority are inactive. The inactive units outputs zero values meaning that their weights aren't updated either during the pass. This forces the network to learn more robust features with less neurons available. This has been shown generally to improve the networks performance and reducing overfitting with the cost of increased training time [Sri+14; GG16]. The inactive units are chosen randomly for each pass.

The model aims to break the n-dimensional data to higher dimensions and then proceeds to reconstruct the input data. The recreated data is then compared with the input data. An anomaly is defined here as an event that the LSTM model cannot predict with great uncertainty. The network consists of input, output and 3 hidden connected LSTM layers, thus creating a stacked LSTM network with dropout layers. The first layer is of size 128, the middle layer is of size 256 and the last hidden layer is of size 64 and all neurons are using hyperbolic activation function $\sigma(x) = \tanh(x)$. The model is trained to recreate the input data therefore the quadratic loss function is calculated as $L = \sum^{n}(X - \hat{y})^2/n$, where the $\hat{y}$ represents the models attempt to reconstruct the data. Model is trained with

50000 log messages for 50 epochs with sequences of 100 logs and batch size of 32. Thus 3200 log vectors are backpropagated every time before updating the weights.

5. Results and evaluation

In this chapter the embedding models are compared with each other.

To measure the descriptiveness of different embedding models is a difficult and somewhat unclear task. How to determine whether one model's embedded vectors are better describing the log data than other model's? Perhaps the simplest way is just to run the data through the anomaly detection models and rate their success.

However, before the anomaly detection review I will take the Doc2vec model into further inspection. Measuring the distance between two vectors, one can see that the model does not distinguish two messages from each other well. From the following messages the first two are identical but occurred in different time step and the third is completely different message and none of the words are same as in the first two messages.

1. `[thread-1] DEBUG DBUtil SELECT sequencevalue FROM sdcsequence WHERE sdcid = ? AND sequenceid = ?`
2. `[thread-1] DEBUG DBUtil SELECT sequencevalue FROM sdcsequence WHERE sdcid = ? AND sequenceid = ?`
3. `[thread-2] INFO ActionService About to go through 1 command(s).`

Calculating the Euclidean distance between the messages we get $d(1,2) = 0.0024$, $d(2,3) = 0.0020$ and $d(1,3) = 0.0006$. Thus, the first message is closer to the third than the second message even though the first and second messages are identical. The same problem does not occur with the other embedding models since the log vectors are created deterministic so that identical log messages gets identical log vectors. Though there's a chance that two different log messages would get the same log vector.

Note that this evaluation does not yet explain the success or failure of the embedding in terms of anomaly detection, it just explains that the model has not reached its goal explaining the log message with respect to the other log messages.

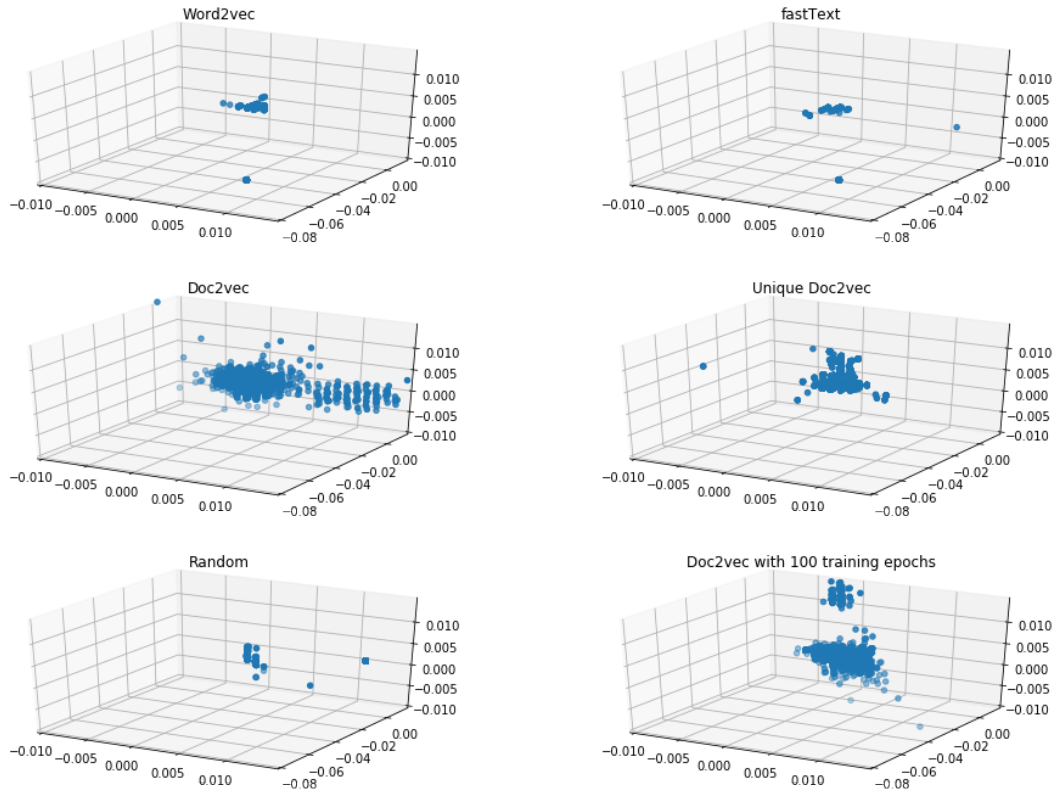

Figure 5.1: Plotting of 5000 log vectors projected from 100 dimensions to 3 with ICA. The sixth plot is from Doc2vec model with 100 training epochs instead of 20. However, it is log vectors sill did not converge significantly closer towards similar logs.

5.1 Independent Component Analysis

Here is shown the results of dimension reduced data with the different models. Looking at the figures 5.1 and 5.2 one can easily see the difference between the doc2vec and the other models. The doc2vec models projects the data to cloud of points and the other models have much more dense output.

For all of the embedding models it seems that longer log messages are regularly projected further away from the other messages. This might be troublesome regarding anomaly detection since longer messages are not anomalous by default. Many of the longer messages contains some SQL queries which in many cases makes the log message to be longer than 25 words.

The table 5.1 shows some metrics of the clusters. From this table we can see that the standard deviation of the cluster size is much lower for the Doc2vec and *unique Doc2vec* models. This implies that cluster are not as dense as in the other embedding models. The metrics also implies that the clusters from Word2vec,

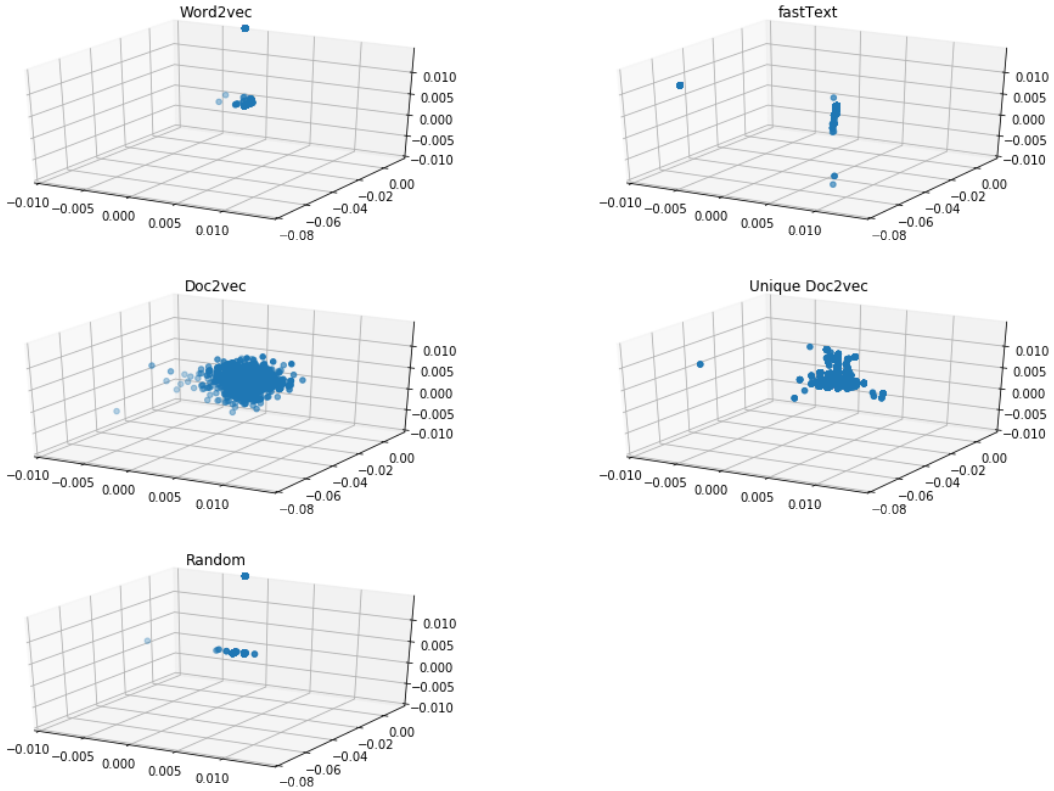

Figure 5.2: Plotting of 5000 log vectors projected from 300 dimensions to 3 with ICA.

Embed. model	Standard deviation	Largest cluster size	Less than 100 messages	More than 15 000 messages
w2v 100	12677.318	80 216	29	8
fast100	13282.093	87 844	33	5
d2v 100	3825.317	14 094	8	0
ud2v 100	4762.622	18 279	2	5
Random 100	10785.136	68 362	25	7
w2v 300	12586.662	77 434	28	6
fast 300	12223.719	78 072	32	8
d2v 300	3905.764	15 173	8	1
ud2v 300	5071.340	20 193	2	6
Random 300	11518.842	71 549	22	8

Table 5.1: Table shows metrics from each embedding model's cluster size.

fastText and random vector embedding are considerably uneven. That means only a few of the clusters contains the majority of the messages. For many other clustering cases this might be a problem, but in this case must be noted that the original log data is also uneven. Thus, the uneven clustering can be seen as asset.

Finally let's take better look at some more frequent messages.

1. `[thread-1] INFO ActionService About to go through 1 command(s).`
2. `[thread-1] INFO ActionService Processing action block`
3. `[thread-1] DEBUG SecurityService Getting Connection for connectionid <id-1>`

Similar or same messages appear also in dozens of other clusters in the Doc2vec and *unique Doc2vec* models. Same observation could be made with the other embedding models but on much fewer cases and not as severe. For example the third log here is found in 2 clusters for Word2vec and fastText models, in 5 clusters for random embedding model and over 60 clusters for Doc2vec and *unique Doc2vec* models. It seems that the messages with a changing parameter, as connection id, are clustered better for Word2vec and fastText than in random vector embedding models.

5.2 Hidden Markov Model

The figures 5.3 and 5.4 shows the probabilities of log sequences of size 20 and 100. The predictions of word2vec, fastText and the random embedding are fairly similar with each other and the Doc2vec and *unique Doc2vec* are similar to each other. The doc2vec based models give fairly homogenous plots without much fluctuation with the probability. However, the Word2vec, fastText and random embedding models gives plots from which there can be distinguished a downward spike at index 5525. The spike is easier to distinguish in the figure 5.3 than in figure 5.4 but it can be seen in both. It is also more distinguishable in the 300-dimensional models. The figure 5.5 shows how the states change during the spike and Word2vec, fastText and random embedding models seem to show that during the spike there is clearly only one hidden state active. Another interesting part is that around the index 8200 there is a sequence that has high and very stable probability. The figure 5.6 shows how the hidden states change around these time steps. Interestingly this part is also distinguishable in the *unique Doc2vec* model and a bit in the Doc2vec model.

Taking closer look at the log messages around downward spike we can see that there are two error messages around it. The errors are as following:

```
[thread-1] ERROR DateTimeService Can't parse date from '' with format 'FastDateFormat[dd.MM.yy
HH:mm,Europe]'.
java.text.ParseException: Unparseable date: ""
at java.text.DateFormat.parse(DateFormat.java:366)
at com.softwarepoint.services.DateTimeService.getLocalDateTimeFrom(DateTimeService.java:297)
at com.softwarepoint.services.DateTimeService.getLocalDateTimeSilently(DateTimeService.java:320)
...
```

The error is about trying to parse a date from empty string and the error is probably cause of the code not handling an empty string. However, the same error does occur couple other times and some of the cases the model fails to highlight it. It could be that the error itself isn't that unlikely to occur and the downward spike is highlighted only given the circumstances.

Taking closer look at the high and stable probability spike there is a recurring series of messages found such as:

```
[thread-1] DEBUG SecurityManager END: getConnection

[thread-1] DEBUG SecurityManager START: getConnection

[thread-1] DEBUG SecurityService Getting Connection for connectionid '<user>'
```

This usually is a sign of user being idle and the server is polling the client. Enlightened guess is that many of the users have left for lunch at the time since the timestamp is close to 11.00.

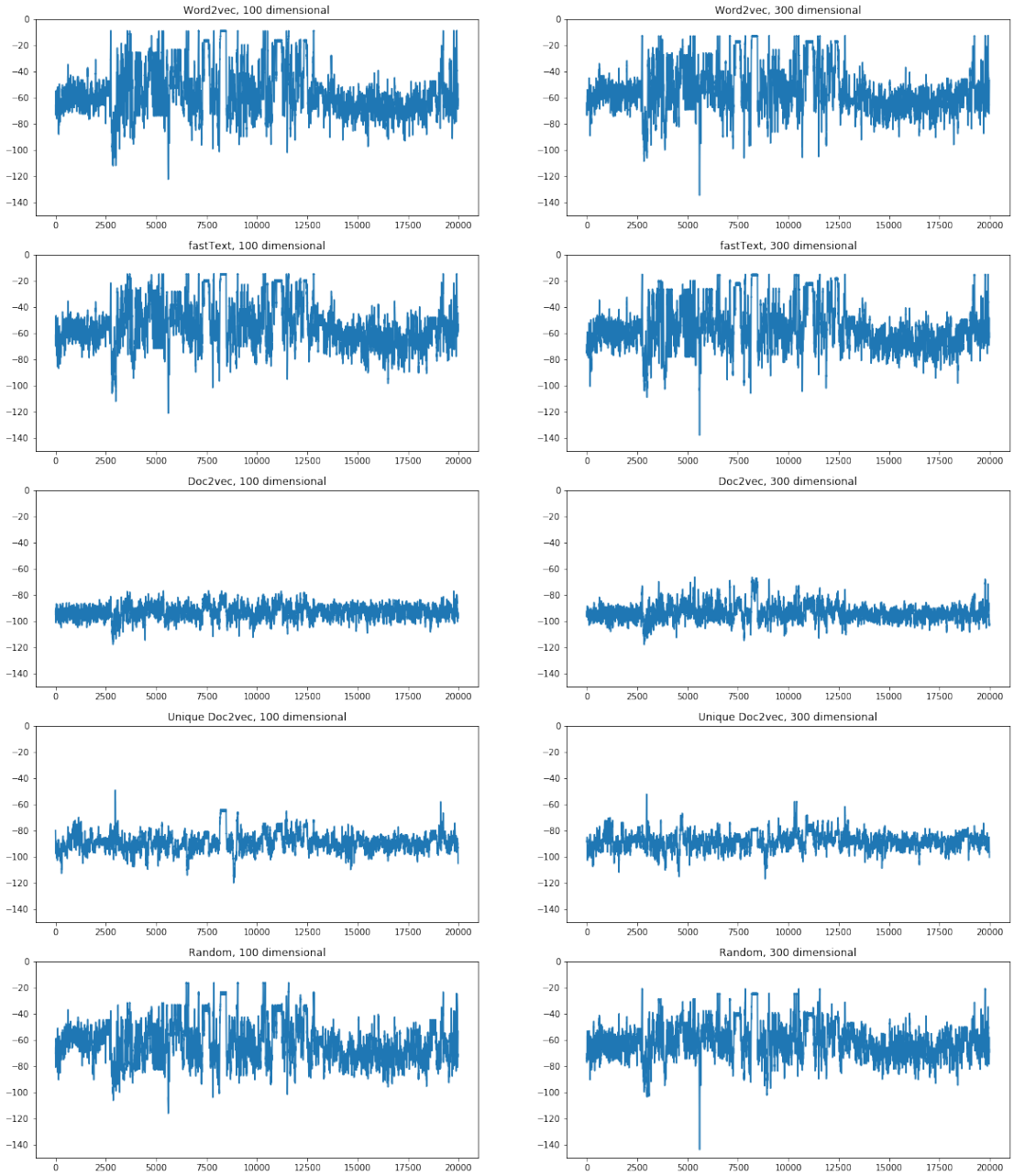

Figure 5.3: The probabilities of log sequences calculated with Viterbi algorithm and sliding window size is 20 log messages. The x-axis shows index of the middle log in the sequence and y-axis is the negative log probability for the log sequence

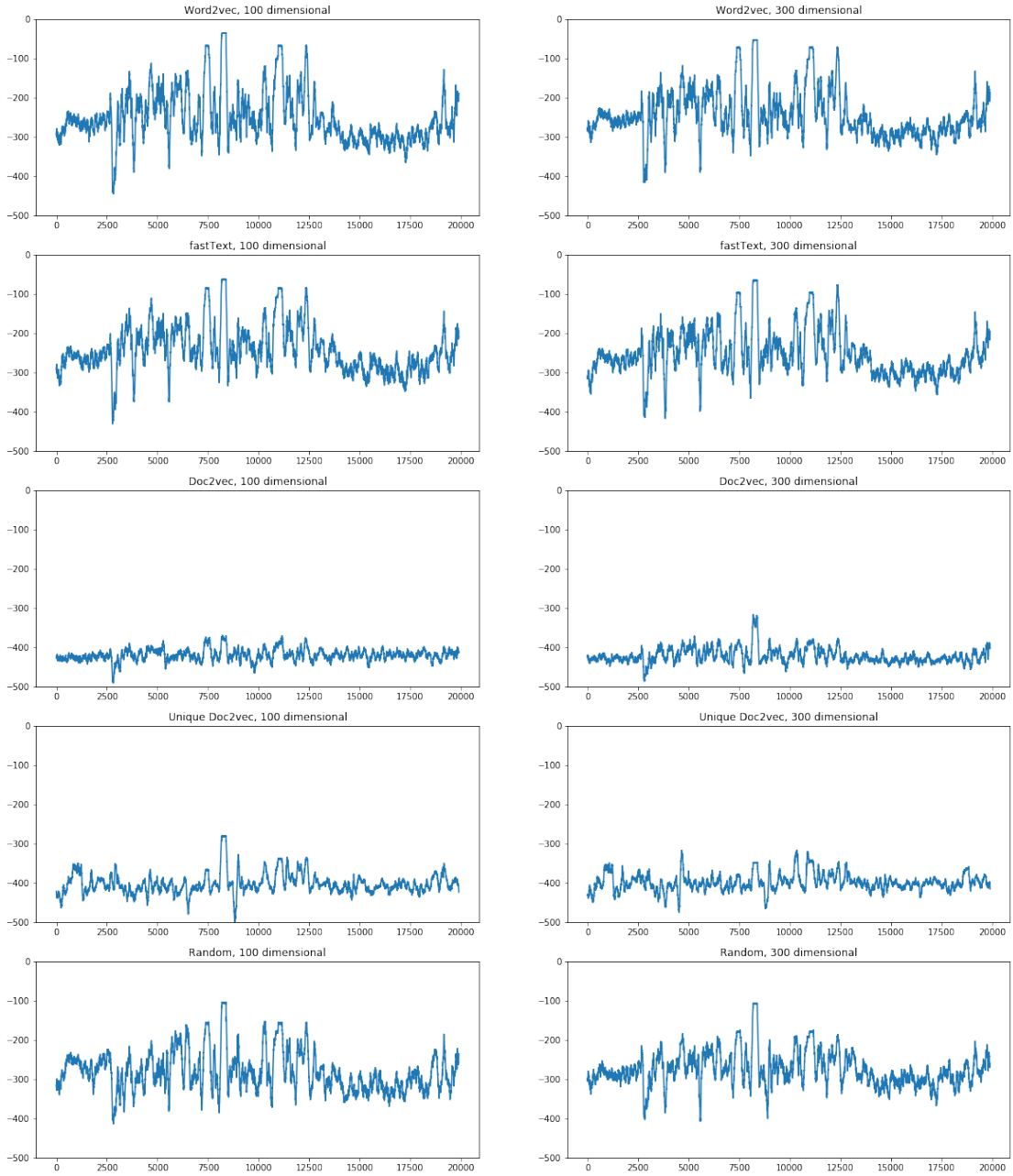

Figure 5.4: The probabilities of log sequences calculated with Viterbi algorithm and sliding window size is 100 log messages. The x-axis shows index of the middle log in the sequence and y-axis is the negative log probability for the log sequence

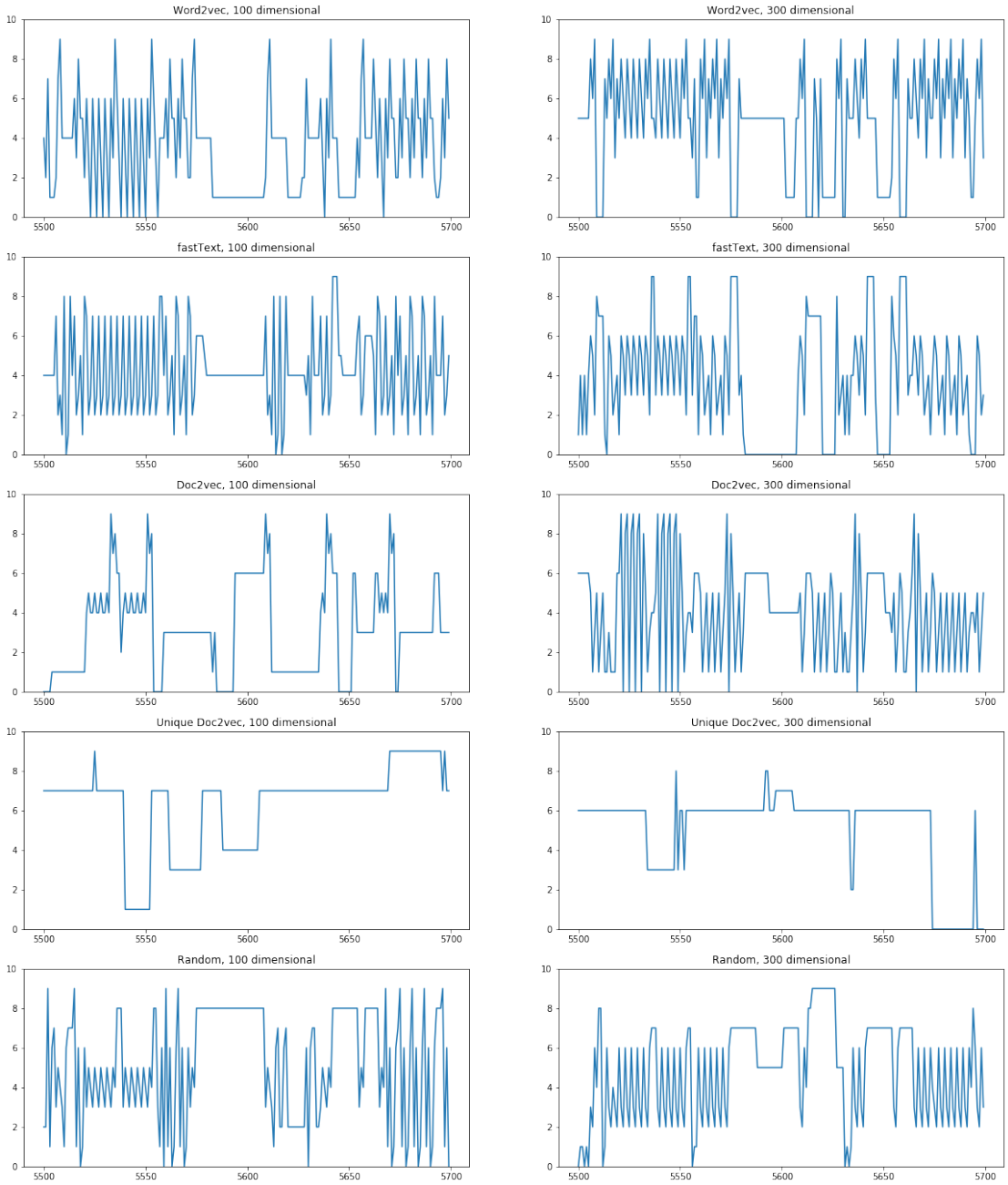

Figure 5.5: Plot shows how the predicted hidden state changes. The interval is chosen around the part which has low probability in figure 5.3 and fairly low in figure 5.4. The x-axis shows index of the middle log in the sequence and y-axis is the predicted hidden state.

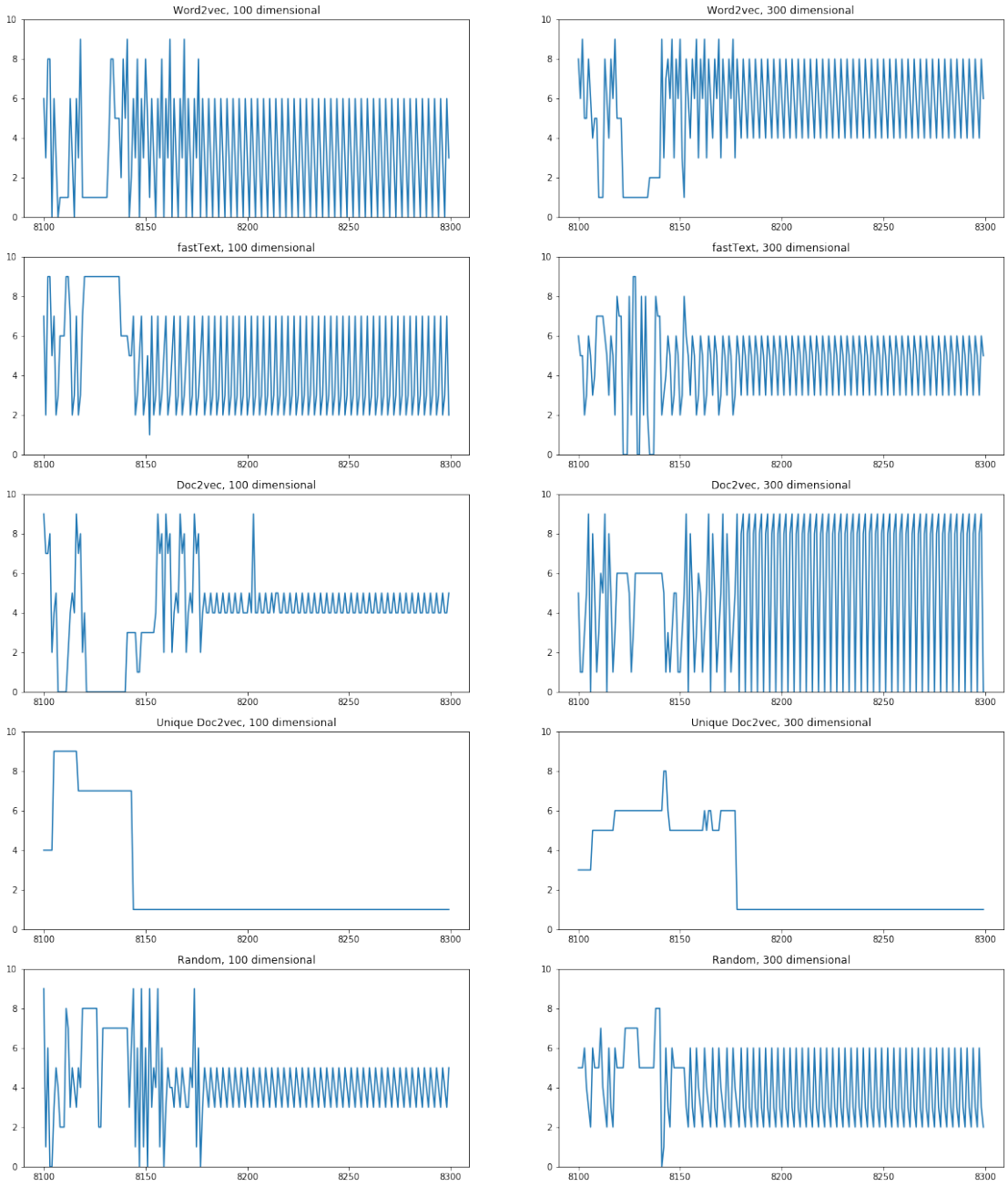

Figure 5.6: Plot shows how the predicted hidden state changes. The interval is chosen around the part which has steady probability in figures 5.3 and 5.4.

5.3 Long Short-Term Memory

From the training and validation loss graphs 5.9 it can be seen that the model struggles to lower the loss with Doc2vec and *unique Doc2vec* models comparing to the other embedding models. This is indication of the poorly converged log vectors and that the log vectors given by Doc2vec seem randomly distributed without easily detectable patterns. The other models seem to find some converging but probably with longer training time all of the models could improve. Perhaps surprisingly the random vector embedding is converging almost as well as the Word2vec and fastText models.

From the figures 5.7 and 5.8 we can see that the results are again quite similar regardless of the dimensionality used for the embeddings. Also, the effect of the ICA can be seen in the input data graphs of Word2vec, fastText and random embedding, when the separated sources are show discrete values instead of three continuous curves. The auto-encoders trained with Doc2vec based embeddings does not seem to produce sensible results as the training loss graph suggests. The other models seem to produce similar plot as the input but it is noticeable that none of the models can predict the spikes in the middle of the input data. The highlighted messages are as follows:

```
[thread-1] INFO RequestController [Cookie: Portal=AD570CA3B12EA6C9F5095C1858A088CF9AC...]
```

Essentially the log message is a very long message which is just parsed badly so that the created vector is projected further away from most of the log vectors. Since the message itself is quite rare the LSTM fails to predict it. Also, the LSTM autoencoder doesn't seem to highlight same messages as the HMM. To get better results from the LSTM model it would need to be trained more. 50 epochs does not seem to be enough even for the already flourished Word2vec or fastText models. However, at this point it could be said that the Doc2vec based models does not work as well as the other models with the LSTM anomaly detection.

Figure 5.7: Graph shows the results of LSTM Autoencoder when using 100 dimensional embeddings as input data. The left column shows how the input data looks, the middle column shows the autoencoded result and the right column shows the absolute error.

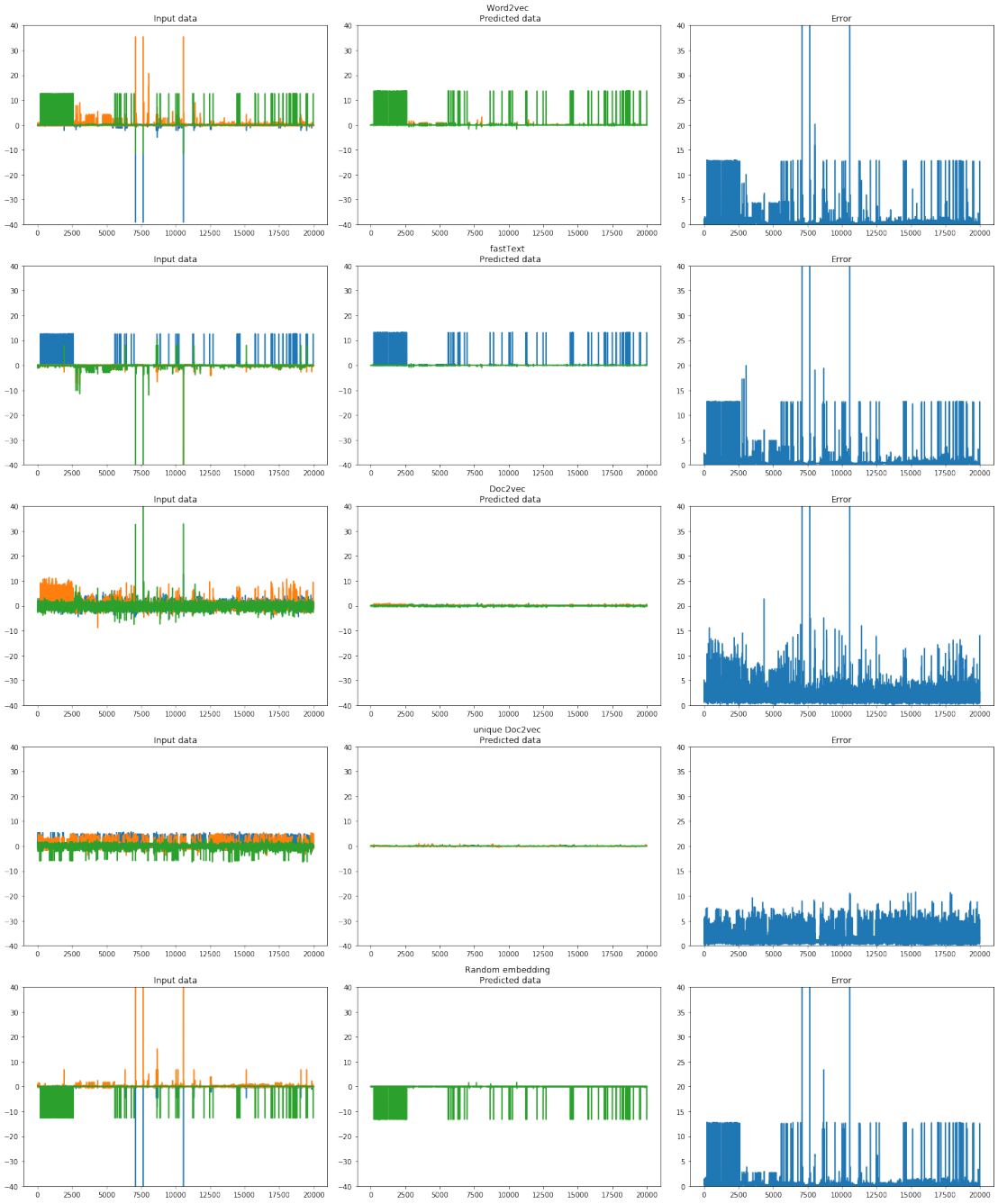

Figure 5.8: Graph shows the results of LSTM Autoencoder when using 300 dimensional emebeddings as input data. The graphs look quite similiar to the results from 100 dimensional embeddings and both results highilght the 3 error spikes at the indexes 7199, 7756 and 10671.

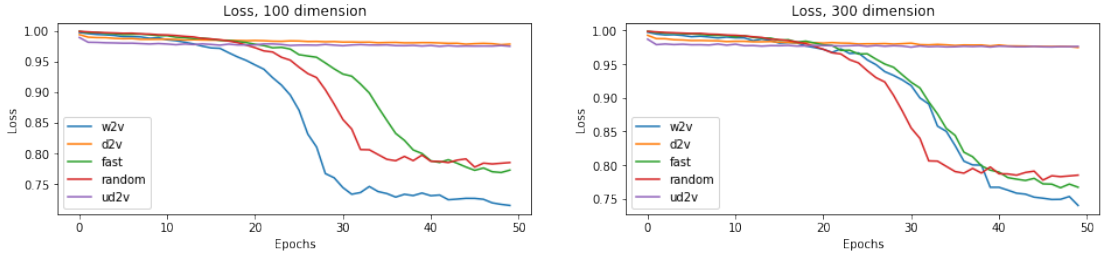

Figure 5.9: The graph shows the loss of the LSTM model between each epochs. It seems that there is little or no effect on the training when using word embeddings with 100 or 300 dimensions. The Doc2vec models does not seem to converge at all during the training whereas the other models improve for some extent. However, training the models further would probably decrease the loss for each model.

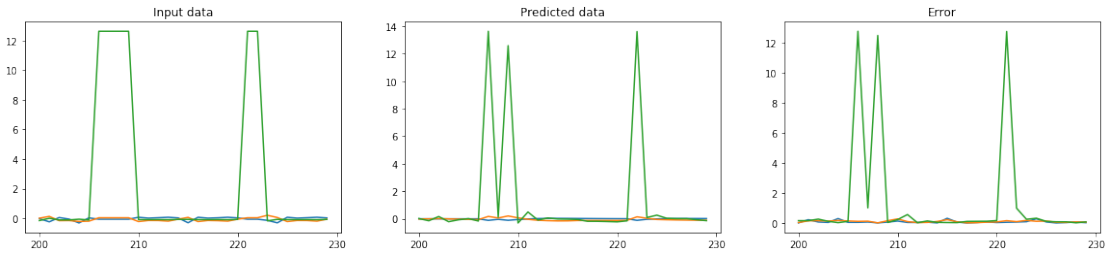

Figure 5.10: Closer look at Word2vec data with LSTM prediction and error. The LSTM can predict the spikes in green signal only partly. Longer training times for the LSTM could make the prediction more precise and the error plots would be easier to interpret.

6. Discussion

6.1 Embedding models

There was minimal difference between the results of 100 and 300-dimensional log vectors. However, taking the computational toll that 300 dimensional embeddings brings I would recommend using 100 dimensional embeddings instead. The training time of the embedding models was not so different but simply saving and loading log vectors that are three times bigger takes more time and effort at least when working with bigger datasets.

For anomaly detection purposes the Word2vec and fastText embeddings seemed to produce best results with the least training time. Training the anomaly detection models with Doc2vec's embeddings took roughly five times the time compared to other models. The log vectors from Doc2vec and *unique Doc2vec* were too randomly distributed and for the anomaly detection models it was difficult to find repeating patterns. Even though the unique Doc2vec created vectors for each unique log message, the clusters from these vectors didn't seem to include similar messages as in the clusters from Word2vec and fastText embedding models. The LSTM model could probably create moderate results from the Doc2vec and unique Doc2vec but they would require longer training times and still it would trail the results from Word2vec or fastText. Perhaps suprisingly the random vector embedding succeeded in HMM and LSTM quite well, even though it had the same problem as the Doc2vec based models and had clusters with mixed messages.

6.2 Anomaly detector

Anomaly detection with just the ICA model and clustering did not work well since the outliers were usually just longer log messages. This is not because the actual ICA model didn't work well for its task but instead how the log vectors were constructed by summation of word vectors. However, the ICA's biggest utility was to reduce the dimensionality of the log vectors so they can be clustered and analysed with the other models.

The HMM and LSTM could produce better results even when the longer log messages were problematic. However, each of the highlighted message is still needed to be validated as anomaly manually and at current state the anomaly detector will probably create more work for system administrators instead of reducing it. Thus, the produced anomaly detector is not yet viable for automatic system monitoring since its prone to point out many false positives as anomalies. However, I can see that the detector could be used by development when fixing already reported bugs which are difficult to reproduce or detect.

7. Conclusion

I presented three word embedding models which can be used to give vector embeddings to event logs. In addition, two variations for these models were introduced and in total five different embedding models were reviewed. Each of these models created 100 and 300-dimensional vector embeddings of event logs which were then further analysed with three anomaly detection models.

The results of this thesis show that Word2vec and fastText word embeddings surpass the Doc2vec style of paragraph embedding when trying to analyse event logs. If the training time of the embedding models is much higher than in this thesis, then the case might be different. The anomaly detection models still need some improving to gain more useful information but the HMM and LSTM model could generate useful tool for software development and perhaps in some cases for system administrators also.

7.1 Future work and improvements

Perhaps the biggest problem with the given anomaly detection solutions is that the longer the message is the more likely it is considered as an anomaly. This could be solved by changing the style how the log vector is constructed. In the current Word2vec and fastText solutions each word in the log message gets an embedding which are summed together with the other word vectors from the log message. If a message has a lot of words, it will get more summations which will set the log vector further away from the other log vectors. This could be solved by dividing the summation with the amount of words in the log message or some similar methods.

The study with rather fixed values for the embedding models and only the different embedding models were on focus. It could have significant effect for anomaly detection by fine tuning some of the embedding or anomaly detection model parameters such as the window width in the embedding models. For the Word2vec and the fastText it might give better descriptiveness of the data if the window width would be set much longer so that it can inspect the words in the next log message as well. Also, instead of using arbitrary amount of 100 log messages

for each training sequence, the length of a sequence could be tailored to be 1 minute intervals of logs or all logs from certain connection id. This would make each training sequence different length, but the sequence would be more natural feed of logs.

This study was made only on one server's logs. In theory the models work generally for any textual log messages and the anomaly detector should produce similar results in different applications, but it would need to be tested. Also, additional embedding methods could be introduced for comparison with the ones seen in this study. The additional embedding methods could be including but not limited to GloVe and Latent Dirichlet allocation.

7.2 Acknowledgements

In the end I would like to show my gratitude towards my supervisor Seppo Nyrkkö for the necessary guidance and Software Point for the opportunity to work on this project.

Bibliography

[For73] G David Forney. "The viterbi algorithm". In: *Proceedings of the IEEE* 61.3 (1973), pp. 268–278.

[RJ86] Lawrence R Rabiner and Biing-Hwang Juang. "An introduction to hidden Markov models". In: *ieee assp magazine* 3.1 (1986), pp. 4–16.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[Hyv99] Aapo Hyvarinen. "Fast and robust fixed-point algorithms for independent component analysis". In: *IEEE transactions on Neural Networks* 10.3 (1999), pp. 626–634.

[WFP99] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. "Detecting intrusions using system calls: Alternative data models". In: *Proceedings of the 1999 IEEE symposium on security and privacy (Cat. No. 99CB36344)*. IEEE. 1999, pp. 133–145.

[HO00] Aapo Hyvärinen and Erkki Oja. "Independent component analysis: algorithms and applications". In: *Neural networks* 13.4-5 (2000), pp. 411–430.

[MHA00] Kerstin M.L. Menne, Henning Hermjakob, and Rolf Apweiler. "A comparison of signal sequence prediction methods using a test set of signal peptides ". In: *Bioinformatics* 16.8 (Aug. 2000), pp. 741–742. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/16.8.741. eprint: http://oup.prod.sis.lan/bioinformatics/article-pdf/16/8/741/470169/160741.pdf. URL: https://doi.org/10.1093/bioinformatics/16.8.741.

[CZ05] Pimwadee Chaovalit and Lina Zhou. "Movie review mining: A comparison between supervised and unsupervised classification approaches". In: *Proceedings of the 38th annual Hawaii international conference on system sciences*. IEEE. 2005, pp. 112c–112c.

[YM05] Kenji Yamanishi and Yuko Maruyama. “Dynamic syslog mining for network failure monitoring”. In: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM. 2005, pp. 499–508.

[Cul10] Aron Culotta. “Towards detecting influenza epidemics by analyzing Twitter messages”. In: *Proceedings of the first workshop on social media analytics*. acm. 2010, pp. 115–122.

[LZL10] Z. Lan, Z. Zheng, and Y. Li. “Toward Automated Anomaly Identification in Large-Scale Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 21.2 (Feb. 2010), pp. 174–187. ISSN: 1045-9219. DOI: 10.1109/TPDS.2009.52.

[MYZ13] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. “Linguistic regularities in continuous space word representations”. In: *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2013, pp. 746–751.

[Mik+13] Tomas Mikolov et al. “Distributed representations of words and phrases and their compositionality”. In: *Advances in neural information processing systems*. 2013, pp. 3111–3119.

[LM14] Quoc Le and Tomas Mikolov. “Distributed representations of sentences and documents”. In: *International conference on machine learning*. 2014, pp. 1188–1196.

[Sri+14] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[LLJ15] Jiwei Li, Minh-Thang Luong, and Dan Jurafsky. “A hierarchical neural autoencoder for paragraphs and documents”. In: *arXiv preprint arXiv:1506.01057* (2015).

[Boj+16] Piotr Bojanowski et al. “Enriching Word Vectors with Subword Information”. In: *CoRR* abs/1607.04606 (2016). arXiv: 1607.04606. URL: http://arxiv.org/abs/1607.04606.

[GG16] Yarin Gal and Zoubin Ghahramani. “A theoretically grounded application of dropout in recurrent neural networks”. In: *Advances in neural information processing systems*. 2016, pp. 1019–1027.

[He+16] Shilin He et al. “Experience report: System log analysis for anomaly detection”. In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2016, pp. 207–218.

[Mal+16] Pankaj Malhotra et al. “LSTM-based encoder-decoder for multi-sensor anomaly detection”. In: *arXiv preprint arXiv:1607.00148* (2016).

[PC16] Maximilian Panzner and Philipp Cimiano. “Comparing hidden markov models and long short term memory neural networks for learning action representations”. In: *International Workshop on Machine Learning, Optimization, and Big Data*. Springer. 2016, pp. 94–105.

[Gre+17] Klaus Greff et al. “LSTM: A Search Space Odyssey”. In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10 (Oct. 2017), pp. 2222–2232. ISSN: 2162-2388. DOI: 10.1109/tnnls.2016.2582924. URL: http://dx.doi.org/10.1109/TNNLS.2016.2582924.

[SAA17] Dima Suleiman, Arafat Awajan, and Nailah Al-Madi. “Deep Learning Based Technique for Plagiarism Detection in Arabic Texts”. In: Oct. 2017. DOI: 10.1109/ICTCS.2017.42.

[Xu+17] Weidi Xu et al. “Variational autoencoder for semi-supervised text classification”. In: *Thirty-First AAAI Conference on Artificial Intelligence*. 2017.

[Mim19] Mamoru Mimura. “An Attempt to Read Network Traffic with Doc2vec”. In: *Journal of Information Processing* 27 (2019), pp. 711–719.