Bachelor's Thesis

Finding Semantic Units in Deep Language Models

Jasmin Denk

Examiner: Prof. Dr. Hannah Bast
Adviser: Prof. Dr. Hannah Bast

Albert-Ludwigs-University Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Algorithms and Data Structures

June 26th, 2018

Writing period

26. 03. 2018 – 26. 06. 2018

Examiner

Prof. Dr. Hannah Bast

Adviser

Prof. Dr. Hannah Bast

Declaration

Abstract

In this thesis, we want to reproduce the results of the paper "Learning to Generate Reviews and Discovering Sentiment" [Radford et al., 2017]. The authors achieved extraordinary results on sentiment analysis by training a neural language model on millions of product reviews to capture the concept of sentiment. Not only did their model achieve this task, it also evolved a single unit which is responsible for analysing the sentiment of the read data.

Our goal is to implement our own language model and analyse its capability to evolve such a semantic, concept-containing unit following the approach described in the paper. We are also interested to observe the extent to which the size of the training data influences the overall performance. In addition to product reviews, we train our language model on lyrics and emails. We want to see if a language model can also learn to recognize different semantic concepts, such as the mood of lyrics or to classify if an email is spam, with one single unit. While we do not quite reach the results of the paper, our model's best results approximates them. We also analyse the existence of such specific semantic units in other areas of text analysis. Furthermore, we describe a process with which we can identify potential semantic units in language models.

Zusammenfassung

In dieser Arbeit wollen wir die Ergebnisse des Artikels “Learning to Generate Reviews and Discovering Sentiment” [Radford et al., 2017] reproduzieren. Die Autoren erreichen sehr gute Ergebnisse in der Stimmungsanalyse, indem sie ein neurales Sprachmodell mit Millionen von Produktrezensionen trainieren, sodass es das Konzept der Stimmung versteht. Dieses Modell erfüllt nicht nur diese Aufgabe, sondern es bildet auch eine einzige Einheit heraus, die für das Analysieren der Stimmung des gelesenen Textes verantwortlich ist.

Unser Ziel ist es, ein eigenes Sprachmodell zu implementieren und seine Fähigkeit zu untersuchen, semantische, konzept-enthaltende Einheiten herauszubilden, wie in dem Artikel beschrieben wurde. Uns interessiert auch, inwiefern die Größe der Trainingsdaten das Ergebnis beeinflusst. Zusätzlich zu Rezensionen trainieren wir unser Sprachmodell auch mit Liedtexten und E-Mails. Wir wollen sehen, ob ein Sprachmodell verschiedene semantische Konzepte mit einer einzigen Einheit erkennen kann, wie beispielsweise die Stimmung eines Liedtextes, oder ob eine Nachricht eine Spam-E-Mail ist. Während wir die Ergebnisse des Artikels nicht ganz erreichen, kommt unser eigenes Modell diesen nah. Wir analysieren zusätzlich die Existenx solcher semantischen Einheiten in anderen Textanalysefeldern. Außerdem beschreiben wir ein Prozess, mit welchem wir solche potentiellen semantischen Einheiten in Sprachmodellen identifizieren können.

Contents

1 Introduction

Determining the sentiment of a writer in respect to a certain topic has caught the interest of the scientific community, but also sees applicability in other areas. The business world can also profit from knowing what the customers prefer or dislike [Cambria et al., 2013]. Even the sentiment in financial news articles can help to predict the future stock market [Lubitz, 2017]. In this thesis, we deal with the task of classifying the polarity of a given text. That is to find out whether the expressed opinion in this given text is positive or negative.

One of the main challenges in developing machine learning algorithms that analyse sentiment is to convert these natural language expressions to a more processable form. How a given text should be best represented to capture all important features is a widely researched topic [Bengio et al., 2013]. We approach sentiment analysis by focusing on this task of data representation.

Following the lead of [Radford et al., 2017], we train a neural language model to represent and comprise the most characterizable features of a given text. Subsequently, these learned representations should contain the concept of sentiment when the sentiment of the given training text is an important feature. We will elaborate this intuitively over the next few paragraphs.

A neural language model tries to predict the next natural language instance (sentences, words, characters, bytes) given a set of previous instances. We implemented a character-level language model, such that it tries to predict which character most likely follows a sequence of previous characters. By comparing the predicted character to the actual next character and repeating this process consistently for the next sequence of characters, the model learns and improves its prediction. This kind of learning, where a model does not need to know the label ("positive" or "negative") of the given training text, is called unsupervised learning. In this case, the dataset is called unlabelled. In contrast, models which need to know the associated label of a given example perform supervised learning. They have to train on labelled datasets.

To improve the guessing of the next character, the used neural language model learns to best represent the essence of what it has already read. For simplicity, let's say that this representation is encoded as a vector called the cell state. This vector represents the condensed summary of what the model has read before, and is updated after every read character. Based on this summary, the model predicts the next character.

As an example, the given text "I hated this book! It was " will certainly continue differently than "I loved this book! It was ". A good language model should have learned to identify such differences while training and predict different characters for each of these starting sentences. A compilation of how three of our trained language models predict how the sentences are continued can be seen in Table 1.1.

When training the language model on a large dataset of product reviews, it is crucial for the model to recognize the sentiment of the data to accurately predict the next characters. This means that the learned representation vector of processed data, the cell state, has to perform sentiment analysis.

starting text	predicted continuation
I hated this book! It was	a waste of time and money.
	so boring and the characters were so bad that I couldn't even finish it.
	a little difficult to read and the story was so bad that I was really disappointed in the story.
I loved this book! It was	a great read and I would recommend it to anyone who likes a good romance.
	a great read and I couldn't put it down.
	a great read and I was so excited to read the next book in the series.

Table 1.1: Example text generation from different language models given two starting texts. The predicted next character was treated as the actual next character to let the language models continue the sentence.

[Radford et al., 2017] implemented and trained such a language model on millions of product reviews, achieving very good results on multiple sentiment datasets (see section 2). They observed that one unit, which is responsible for exactly one value in the cell state of their trained model, seems to be directly responsible for the good results in sentiment classification. They called this unit their sentiment unit.

We aim to recreate their results by training our own implemented neural language model on the same data. Furthermore, we evaluate it on the same sentiment datasets using a similar supervised classifier. To further analyse what influences the results, we train our language model on different subsets of data and while using different hyperparameters. In addition, we are interested to see if other "high-level concepts" [Radford et al., 2017, p. 1] can be learned and represented by a neural language model. Therefore, we widen our line of research and test our approach on other classification tasks by training our language model on lyrics and emails.

Although it does not reach the 91.8% achieved in [Radford et al., 2017], our best trained model approaches this value, reaching a maximum accuracy of 87%. We were also able to find our own sentiment unit responsible for this good result.

What follows now is a quick overview of our approach.

In this thesis, we describe a system which is a generalized version of the methodology presented in [Radford et al., 2017]. This system contains a neural language model and a linear classifier, and can be used to identify a potential semantic, concept-containing unit in the trained language model.

We start by choosing a semantic concept we want to train on. The focus here is on sentiment analysis. After pre-processing the training data for our language model, we train it given a set of chosen hyperparameters. Afterwards, we evaluate how good the model has encapsulated the concept we are looking for. For this, we use a labelled sentiment dataset and let our trained language model process the containing examples. After processing the, the language model returns the respective representation of these examples. We use the representations of these sentences as features for our linear classifier. After training this classifier, we inspect which feature has the most influence for correctly classifying the sentiment. The unit of the language model associated with this feature is a potential semantic, concept-containing unit. To test to what extend this unit encapsulates the learned concept, we train the classifier again. This time, we use only this particular unit as feature. To further analyse what exactly influences the result of the whole model, as well as the particular concept-containing unit, we try out different parameters and train on different subsets of the training data.

The remainder of this thesis is structured in the following way. In the next section, we will present the approaches of other authors on sentiment analysis, which also focused on data representation. In section 3, we introduce the main components of our system: the language model as broadly described above and the linear classifier used to evaluate the learned representations. In section 4, we present how the before mentioned components interact in the precess to find a concept-containing unit. Section 5 contains the evaluation, where we describe our main results on sentiment analysis as well as spam and mood classification. We summarise our findings in section 6.

2 Related Work

In this section, we present three recent approaches on the task of sentiment analysis. We selected models whose success roots in learning to represent raw data in a more useful form and are therefore suitable comparison examples for our own approach.

The first comparison example is the unsupervised $ParagraphVector$ framework, introduced in [Le and Mikolov, 2014]. The developed model is built upon the CBOW and Skip-gram architecture [Mikolov et al., 2013], derived from word-level neural language models. These architectures are used to learn vector representations of words.

These fixed-size word vectors v_w are trained differently depending on the architecture. Using CBOW, v_w are trained by predicting the word w_i given its context $w_{i-k}, ..., w_{i-1}, w_{i+1}, ..., w_{i+k}$ encoded as their representation $v_{w_{i-k}}, ..., v_{w_{i-1}}, v_{w_{i+1}}, ...,$ $v_{w_{i+k}}$. Using Skip-gram, v_w are trained by predicting the context of a word w_i given its representation v_{w_i}. These v_w correspond to the word embedding of the underlying neural language model. The word vectors are trained to minimize errors in their prediction. After training, semantically similar words have similar vector representations. The introduced $ParagraphVector$ framework extends these models by adding a new trainable matrix. In this matrix, each vector d_p represents a paragraph (or another variable-length entity like a sentence or document) of the training data. While v_w are shared across all paragraphs, d_p are only shared across the respective context.

For their different experiments on supervised datasets, the authors let both models train the paragraph and word vectors on the corresponding training data. Then, they fed a concatenation of the matching paragraph vectors to a classifier. For testing, the word vectors get fixed while the paragraph vectors get extracted and evaluated by the trained classifier. On sentiment analysis datasets, the paragraph vectors achieved very good results with 87.8 % and 92,58 % on a binary and 48.7 % on a multi-class classification task [Le and Mikolov, 2014].

The use of a neural language model to predict words and learning to best represent a given text is a good comparison example to our approach. Two main differences exist. First, we do not use the learned word or phrase embedding as representation but rather another part of our network, the cell state. Second, we train our language model beforehand on a large unlabelled dataset and evaluate the learned representations on related supervised datasets. In contrast, the paragraph vectors are individually trained and tested on each dataset. This makes our model more versatile because, once trained, it can be used to evaluate the learned representations on different datasets. In contrast, the paragraph vectors are more adaptive and need less training data to achieve good results.

The second example of a model focusing on data representation is the *skip-thoughts* model [Kiros et al., 2015]. Given a triple of consecutive sentences $(s_{i-1}s_is_{i+1})$, the authors train an encoder-decoder model. This encoder-decoder model predicts or, more fitting, reconstructs the previous s_{i-1} and next sentence s_{i+1} given the source sentence s_i.

The introduced model consists of one encoder and two decoder networks, respectively implemented with a recurrent neural network (RNN). The encoder takes the source sentence s_i and translates it into a fixed-size vector h, representing the hidden state of the RNN (for details see section 3.1). The decoders take h and predict the next sentence s_{i+1} and the previous sentence s_{i-1} respectively. The encoder has to learn to represent the current sentence as best as possible in order for the decoders to be able to truthfully reconstruct the surrounding sentences.

After training the *skip-thought* model on a large book corpus, the authors extract and evaluate the respective representations of sentences in different supervised dataset using a linear classifier. Tested on sentiment analysis, their model achieved good results between 75% and 80% on different binary sentiment datasets with small variations to their model [Kiros et al., 2015].

This approach resembles our own, especially the fact that we also use a large dataset to train the model before the particular evaluation on a supervised task. However, we do not train to predict the surrounding sentences of an encoded source sentence. Instead, we train to predict the next character given one character and a vector representing the meaning of the previously read characters.

The third model we want to present, *byte mLSTM* introduced by [Radford et al., 2017], is the most relevant point of comparison as we follow the described methodology and want to reproduce their achieved results. This third approach can be seen as a combination of the two previously presented models. By training to predict the next byte on a large text corpus, the network learns to represent the read in text as a fixed-size vector. Once trained, the learned representation can be evaluated on different supervised datasets.

Following the rough sketch in the introduction, [Radford et al., 2017] trained a byte-level neural language model, consisting of a multiplicative LSTM with 4096 units, on approximately 83 million product reviews. The model learns to predict the next byte given mainly the current byte and the previous cell state, as described in section 1.

To evaluate the quality of this learned representation, the authors follow the procedure of [Kiros et al., 2015] and use a linear classifier after extracting the respective representations for phrases of different supervised datasets. Tested on sentiment analysis, the introduced model advances the state-of-the-art to 91.8% and 92.88% on two binary sentiment datasets. Inspecting which unit of the network has the biggest influence on the classification, they discovered a single unit of their model with an outstanding contribution. The authors deduct that this unit is responsible for detecting sentiment in the processed phrase. Using just this single unit, they achieve 91.87 % instead of 92.88%.

As stated before, we aim to reproduce these results by adapting the presented methods. Nonetheless, our execution differs in three main points. First, our implementation of the language model is different (for details see section 3.1). Second, we use different subsets of their used dataset to train our language model to analyse the influence of the dataset size. Last, we not only train our language model on sentiment data but also on data of two additional natural language processing areas, to see if other concept-containing units can evolve in a language model.

3 Components of Our System

In this section, we want to present the two main components of our implemented system. Additionally, we will present background information that is necessary to understand how these components work. These components are the neural language model, presented in section 3.1 and the linear classifier, presented in section 3.2.

3.1 The Language Model

In this subsection, we present the main component of our system, the language model. To give a better understanding of how our implemented neural language model works, we first introduce neural networks in section 3.1.1. There, we also present how these networks are used for language modelling. We go into more details about our implementation of a neural language model in section 3.1.2.

3.1.1 Background: Neural Language Models

In general, a language model is a function that learns to capture the most prominent features of a sequence of words, and assigns a probability to it. In particular, it can be used to calculate the probability of a word given its preceding ones [Bengio, 2008]. We are interested in a character-level language model, so we will talk about a sequence of characters from now on. The probability of a character c_n dependent of t previous ones can be depicted as the conditional probability $\mathrm{P}(c_n|c_{n-t-1}c_{n-t}...c_{n-1})$. There are different methods for calculating this conditional probability.

A neural language model, as introduced in [Bengio et al., 2003], uses a neural network for calculating these probabilities. Neural networks are able to learn distributed representations of data. These distributed representations correspond to a vector of features which characterizes the meaning of the read data. This functionality is inspired by the ability of the brain to generalize characteristics. This ability can be used to recognize similarities between a new object and already known ones [Bengio, 2008].

There are different types of neural networks. In their simplest form, neural networks are known as Feed-Forward Neural Networks (FFNN). The structure described in the following paragraph applies to FFNNs. Other types of neural networks build upon this basic structure.

Neural networks consist of units, also called cells or neurons, divided into three connected layers. The input layer x, consisting of i units, represents given inputs as a i-dimensional feature vector. For language modelling, each character is associated with such a feature vector, for example encoded as an one-hot vector[1]. The hidden layer h, consisting of g units, receives the encoded input. Based on this vector, a trainable $i \times g$ weight matrix W and a trainable g-dimensional bias vector b, the hidden layer calculates its activation. Each unit in h contributes one activation value to this activation vector, the g-dimensional output vector for the next layer. A neural network may consist of multiple hidden layers with different numbers of units. Each of these layers then has its own weight matrix and bias vector to calculate their activation vector based on the respective previous layer. The output layer y, consisting of o units, receives in turn the calculated activations of the previous hidden layer. It calculates its own activation based on a trainable $c \times o$ weight matrix V and a trainable o-dimensional bias vector c. For language modelling, this o-dimensional activation vector is interpreted as the prediction for the next character, matching the encoding of the input layer.

This process of feeding the calculated activation to the respective next layer is called forward propagation or forward phase. During training, the network learns by comparing the predicted outputs to the correct one and hence updating all weight matrices and bias vectors. This process of updating is called back-propagation or update phase.

Language models built upon FFNNs have a disadvantage when dealing with sequential data like text; a character in a consecutive text is dependent on all previously read characters, which cannot be modelled with a fixed-size input vector. For dealing with this problem, a special kind of language model was introduced in [Mikolov et al., 2010]: A Recurrent Neural Network (RNN) based language model.

RNNs are frequently used for natural language processing tasks as they are able to

[1]https://en.wikipedia.org/wiki/One-hot

share previously read information across different timesteps. This is done by storing the activation vector of hidden layer h at each timestep t. This activation vector is used, together with the next character x_{t+1}, W and b, to calculate the new activation of h at timestep $t+1$. This recursively passed g-dimensional activation vector is called the *hidden state* of the RNN. The input and output layer stay unchanged. For language modelling, this means that information about previous characters can persist while the model only has to process one character at a time.

RNNs are trained by unrolling the network with an algorithm called back-propagation through time (BPTT). When unrolling for t timesteps, the RNN can be regarded as t connected FFNNs with shared W and b. In this construction, each hidden layer gets the according input vector x_i and provides the prediction y_i. Weight matrices and bias vectors can now get updated as before for each y_i. For how many timesteps the RNN is unrolled during training is an important factor. It represents how far an input character can directly influence future predictions. As the activation vector is passed to the next sequence of unrollings, a character might be able to have further influence. Nevertheless, dependencies may not be discovered further than this value t.

For better dealing with such log-term dependencies, RNNs with Long Short-Term Memory cells (LSTM) were introduced in [Hochreiter and Schmidhuber, 1997]. A LSTM memory cell can be seen as a wrapping around the original RNN unit which form the hidden layer of the RNN. This wrapping consists of an input gate, an output gate and a forget gate[2] and decides which information will be stored to best represent the previously read text. The LSTM memory cell calculates its activation based on its previous activation vector, the input feature vector x, the weight matrices of the gates and the activation of the wrapped RNN unit. The wrapped unit in turn calculates its activation, as before, based on x and its previous activation. This means that now two states are stored and shared across time: the activation vector of the LSTM memory cells, called *hidden state* as they now form the hidden layer[3], and the activation vector of the wrapped RNN unit, called *cell state*.

[2]The forget layer was first introduced in [Gers et al., 1999]. In [Hochreiter and Schmidhuber, 1997], the wrapping only consists of the output and input gate. As we will be using the implementation with the forget gate, we already introduce it here.

[3]Multiple LSTM cells which use the same gates are called as LSTM memory cell blocks. A LSTM memory cell block of size 1 can be seen as one LSTM memory cell. These blocks are actually the components of the hidden layer. For simplicity, when we speak of a LSTM memory cell, we mean a LSTM memory cell block of size 1.

3.1.2 Our Implementation

We implemented our own neural language model, a RNN using LSTM memory cells, using the framework provided by TensorFlow[4]. Before we are going to describe our implementation, we introduce some important hyperparameters for training neural language models:

- *num_layers* represents the numbers of hidden layers in the network. In our implementation we use one hidden layer.
- *num_units* represents the number of LSTM memory cells and thus the number of wrapped RNN units in our hidden layers. From now on, we use the term "units" when we talk about these wrapped RNN units.
- *num_unrollings* or *seq_length* represents the number of steps the network is unrolled for during training. As stated before, this value is quite important as dependencies between characters might not be recognized further than this number of characters.
- *optimizer* represents which optimization algorithm is used for updating the weights and biases. In our experiments we use the Adam optimizer.
- *learning_rate* influences how much the weights and biases are changed during training. In our implementation we use a learning rate of 0.0005.
- *batch_size* represents the number of text slices we train on in parallel.
- *num_epochs* represents how many times the model iterates over the whole text while training.

Our implemented model first reads in the given training text, slices it in *batch_size* segments and saves it. All unique characters get extracted and used for building two dictionaries: a mapping from each character to an index (called *char-to-id*) and from each index to the character (called *id-to-char*). These dictionaries allow us to transform characters to unique numbers and back. We save the count of unique characters as *vocab_size*. The actual network gets initialized, based on the provided hyperparameters, with a *vocab_size* sized input layer, a *num_units* sized

[4]https://www.tensorflow.org/

hidden layer and a *vocab_size* sized output layer[5]. The units of the input and output layer represent the characters of the vocabulary, according to *char-to-id*, as an one-hot encoded vector. The language model then trains on text segments of size *seq_length* $\times$ *batch_size* by using the hyperparameters described above.

For testing our trained model, we set *batch_size* and *seq_length* to 1 as we do not have to back-propagate. After processing a given start text, the predicted next character gets saved and fed right back as the input to get the next character. This way, we are generating the most probable sequence of characters given the start text.

Furthermore, we implement a function which returns the *cell state*, the activation vector of the wrapped RNN cells in the trained model, after processing a given sequence of characters. This functionality of the trained model is used for the classifier described in section 3.2.2. and further outlined in section 4.

While developing our own language model, we used an implemented LSTM language model from GitHub[6] to get some preliminary results for our experiments. These preliminary results are used for comparison with our achieved final results in section 5.

Comparing our implementation to the one presented in [Radford et al., 2017], we can note some major differences. The authors implemented a byte-level multiplicative LSTM, an advanced LSTM which they observed to converge faster than an ordinary one. They also used the TensorFlow framework but implemented their network (including the LSTM memory cells, the various gates and the layers) from scratch, using the formulae introduced in [Gers et al., 1999]. We cannot say much about further differences as their model is not completely published[7]. According to their paper, they trained their model using the parameters *num_epochs*:=1, *num_layers*:=1, *num_units*:=4096, *batch_size*.=128 , *seq_length*:=256, *learning_rate*.=0.0005 and *optimizer*:=Adam.

[5]The implementation of TensorFlow does not allow us to build a network layer by layer. We instead use a *dynamic_rnn*, consisting of *BasicLSTMcells* to represent the hidden layer and a *fully_connected*-layer to represent the output layer. The input layer gets implemented by choosing the inputs for the *dynamic_rnn*. For simplicity, we assume that we build the LSTM layer for layer.

[6]https://github.com/crazydonkey200/tensorflow-char-rnn (21.5.2018)

[7]The authors published parts of their implementation on GitHub: https://github.com/openai/generating-reviews-discovering-sentiment (20.05.2018). However, they did not include the training and load already trained parameters of the LSTM for testing purposes.

3.2 The Classifier

In this section, we attend to the secondary component of our system: the logistic regression classifier. We introduce the basics of linear classifiers in section 3.2.1. In section 3.2.2, we shortly present our implementation.

3.2.1 Background: Linear Classifiers

The goal of classification in the sense of machine learning is to use characteristics, also called features, of an object and identify which class it belongs to. We will be looking at binary classification; in sentiment analysis, this means that a class can be 0 (negative sentiment) or 1 (positive sentiment). A linear classifier predicts a class based on the features, commonly represented as a feature vector x, a learned weight vector w (with the same dimensionality of x) and a learned bias value b. This prediction can be seen as a decision function $d(x)$ with

$$d(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i * x_i + b > 0, n\text{: number of features} \\ 0 & \text{else} \end{cases}$$

As this sum is a linear combination of the features, these classifiers are called linear. For binary classification tasks, the separation of instances given their classes can be visualized as the hyperplane $w * x = b$, splitting the feature space accordingly. During training, the classifiers try to find the best w and b given x and the label of each training example. This can be seen as an optimization problem, which each classifier solves in a different way [Schütze et al., 2008].

A logistic regression classifier, which we are using in our system, is a linear classifier which uses the standard logistic function $\frac{1}{1+\mathrm{e}^{-t}}$ with $t = w * x + b$ to solve this optimization problem [Yuan et al., 2012]. For evaluation, the probability distribution of the positive class given a feature vector x can be calculated with $P(class = 1|x) = P(d(x) = 1) = \frac{1}{1+\mathrm{e}^{-(wx+b)}}$.

3.2.2 Our Implementation

In our system, we use the implementation *LogisticRegression* from the machine learning library scikit-learn[8]. For training the classifier, we use labelled datasets with

[8]http://scikit-learn.org/

a train/validation/test-split[9]. After reading in the dataset, we create the feature vectors for all instances in the dataset. This is done by loading the previously trained language model and using the implemented function to get the respective *cell state* of the model after processing the given instance. This vector of size *num_units* is used as the respective feature vector for the examples in the dataset. We make sure that the language model does not return falsified representations by receiving text in an unexpected format. This is done by pre-processing the given examples in the same way as the training data for the language model.

While training, we used L1-regularization to ignore irrelevant features [Radford et al., 2017] and try different regularization strengths on the validation dataset to find the best one for the given examples. Afterwards, we evaluate the trained classifier on the test set and document the results.

To be able to compare our learned representations with the results of [Radford et al., 2017], we used the same implemented classifier and procedure for training it.

[9]In some cases, the split is already fully or partially provided by the dataset. If this is not the case, we create it ourself. For details see section 5.

4 The Process of Finding a Semantic Unit

In this section, we will look at how the components introduced in the last chapter interact to find a potential semantic, concept-containing unit in the trained language model. For this, we will walk through the steps we conduct in our experiments, which are presented in the next section. This section is constructed to give an overview of our developed system. For more details on the implementation of our language model and our classifier, we refer the reader to section 3.1.2 and 3.2.2.

Choose a concept and matching data. First, we have to choose a concept which the language model should train to encapsulate. Along with this, we have to find datasets for the language model and the classifier which entail this concept. To train a good language model, we need a large amount of unlabelled text to train on. At the same time, we want to evaluate the learned representations on a related labelled dataset. So before we can start the process of finding a concept-containing unit, we have to fulfil both requirements and find matching datasets, preferably formatted similarly for a truthful evaluation.

Pre-process data for the language model. After choosing the data, we have to prepare the text for training the language model in a certain way. In our use case, we have data consisting of multiple examples (see section 5 for details) and not one continuous text. To show our model where an example starts and ends, we pre-process the data and insert a start token ("\n") and an end token (" ") between the examples, following the methodology of [Radford et al., 2017]. We also strip the example of other newlines, replacing them with whitespaces, and of trailing whitespaces to not falsify the tokens. All pre-processed examples are then written together to a txt-file.

Choose hyperparameters. Before training the language model, we have to decide which hyperparameters to use as they can make a great difference for the trained model. The hyperparameters relevant to our implementation are presented in section 3.1.2.

Start training the language model. We train the language model on the pre-processed file with the chosen hyperparameters as described in section 3.1.2. We save the used hyperparameters and path to the training data in a file for reproducibility and future comparisons. Additionally, we have to save the dictionaries *char-to-id* and *id-to-char* to load them for future use. This is important as the mapping from character and index cannot be recreated and would ruin the trained language model if the relation was lost. We save the trained language model after each epoch.

Optional: Test the language model. To sanity-check our trained language model, we can let it generate text given a start text as described in section 3.1.2. For this step, we need the saved hyperparameters to rebuild the model, the saved dictionaries and the saved model. We may perform this step, especially in the early state of tuning the hyperparameters, to recognize if something went wrong when training the language model. Signs which hint that something is wrong may be that the generated text consists garbled characters or that generated characters are repeated over and over regardless of the start text. When this is the case, we have to check our hyperparameters, training text and potentially the implementation of our model. However, this step is optional as we can perform the next steps to inspect the learned representations.

Train and test the classifier. With the trained language model and the labelled classification dataset, we start training and testing the linear classifier as described in section 3.2.2. The feature vectors used for training the classifier consist of the activation values of the hidden units. Thus, the performance of the classifier depends on the quality of the learned representations in relation to the concept we want to test on.

Search for a concept-containing unit. We inspect the learned weight vector w of the trained linear classifier and determine the highest weight value w_i. We deduct that the feature x_i corresponding to this weight value has the highest significance to the learned decision function $d(x)$ (see section 3.2.1.). This feature corresponds to the activation value of unit i in the learned language model after processing the given example. Thus, we assume that this unit has learned to analyse the given example in relation to the looked for concept, and to represent it in a single value. Following this procure, the sentiment unit was found in [Radford et al., 2017].

To test the quality of this found unit, we train our logistic regression classifier again on the same dataset, but only with the activation of unit i as feature. This means that the classifier has to learn exactly one weight value. The trained classifier is now only dependent on this particular feature. We test the classifier as before and document the results. When the resulting accuracy is good (which has to be analysed individually; i.e. it can be defined as "better than guessing the most probable class"), we have found a unit in our trained language model which corresponds directly to a certain concept. Nevertheless, it is not given that the unit associated with the highest weight value has actually learned the particular concept. This can be seen when the performance of the classifier trained with only this activation value does not approximate the performance when all activations are used. In this case, the language model does not seem to have evolved a concept-containing unit. We call these units, which are associated with the highest weight of the classifier but do not achieve good results, undeveloped concept-containing units.

This process of finding a concept-containing unit can be conducted multiple times with changed hyperparameters and a changed dataset to train the language model on. This is interesting as it allows us to compare the results of the classifier and analyse possible differences. In particular, we are interested to know which hyperparameters influence the results in which way. Additionally, we want to know to what extent the amount of training text matters. The observations we made, together with the documented results, are presented in the next chapter.

5 Evaluation

In this section, we will present our results achieved with the process we introduced in the last chapter. We test our system on three different classification problems with corresponding datasets, representing three different concepts which our language model should learn to represent. These concepts are sentiment in reviews (section 5.1), the integrity of emails (decide if a mail is spam)(5.3) and mood in lyrics (5.2). For each of these topics, we will present the used datasets and our results, also compared to other approaches. In section 5.4, we will discuss the basic complexity of our used algorithms.

For evaluation, we document the f1 score, recall and precision[1] together with the mean accuracy[2]. The first three scores are calculated in regard to the positive class, representing respectively a positive review, a spam mail or a happy lyric. This means that when a classifier always predicts the negative class, independent of the actual labels of the testing examples, all three values are 0.

All language models for the experiments are trained on the cluster of the Chair of Algorithms and Data Structures. The nodes consist respectively of Titan X GPUs with 12 GB VRAM and Intel Core i7-6850K CPUs with 3.6Ghz each. For each of the conducted experiments we use one GPU.

5.1 Sentiment Classification

First, we will present our datasets and results on sentiment analysis. This classification problem is the main focus on our approach; accordingly, these results will be explored and analysed in more detail than the following topics.

[1]respectively implemented with *sklearn.metrics.f1_score*, *sklearn.metrics.recall_score* and *sklearn.precision_score*. See http://scikit-learn.org/stable/modules/model_evaluation.html#precision-recall-f-measure-metrics for more information.

[2]This is the result of the method *score* performed on the trained model given the feature vectors for the testing examples and the correct labels.

Similar to [Radford et al., 2017], we use the Amazon product dataset [He and McAuley, 2016] for training our language model and the binary version of the Stanford Sentiment Treebank (SST) [Socher et al., 2013] for training and testing the classifier[3].

The Amazon product review dataset exists in different versions. As [Radford et al., 2017], we use the aggressively de-duplicated version[4] which contains 82.83 million product reviews, spanning the years 1996 to 2014. Interestingly, the amount of positive reviews heavily outweighs the amount of negative and neutral reviews; the dataset approximately consists of 78% positive, 13% negative and 9% neutral reviews. The source file of this dataset contains additional, for us not relevant information like the reviewer id and the date of the review. We parse this file before pre-processing it (see section 4) so the training data for our language model only consists of the actual text of the reviews. We conduct experiments using three different sized subsets of this parsed dataset to observe how the size of the training text influences the representations learned by the language model. The composition of these subsets can be seen in Table 5.1. For the biggest subset, we decide to use the first occurring 20 million reviews of the original dataset. The composition of this subset corresponds approximately to the composition of the whole dataset.

	composition of subset		
size of subset	positive reviews	negative reviews	neutral reviews
200,000	100,000	100,000	0
2,000,000	1,000,000	1,000,000	0
20,000,000	15,643,930	2,654,532	1,701,538

Table 5.1: Composition of the used Amazon product data subsets. We call a review positive if the respective star-rating is 4 or 5, negative if it is 1 or 2 and neutral if is is 3.

The binary SST dataset comes with a training/validation/test split of 6920 training, 872 validation and 1821 testing examples. As the testing set consists of 909

[3]We use the same pre-processed version as [Radford et al., 2017], using only the text snippets and matching labels without the parse trees which are part of the original dataset. The authors published this version on https://github.com/openai/generating-reviews-discovering-sentiment/data.

[4]http://jmcauley.ucsd.edu/data/amazon/ (12.06.2018)

positive and 912 negative examples, guessing the most probable class would amount to a mean accuracy of 50.1 %. A short excerpt showcasing how the classification examples look like is presented in Table 5.2.

sentence	label
In its ragged, cheap and unassuming way, the movie works.	1
While the film misfires at every level, the biggest downside is the paucity of laughter in what's supposed to be a comedy.	0
I love the way that it took chances and really asks you to take these great leaps of faith and pays off.	1
Lacks heart, depth and, most of all, purpose.	0

Table 5.2: Excerpt from the testing split of the binary SST dataset.

We conduct our sentiment analysis experiments on language models trained on three different magnitudes of training text. In each case, we evaluate six combinations of two different hyperparameters. We train our models for one epoch respectively. The results of said experiments, following the process explained in section 4, can be seen in Table 5.3[5]. When looking at these results, we have to consider that the weight matrices and bias vectors of the language model are randomly initialized at the start of training; small fluctuations can be explained with better or worse starting conditions.

We make three major observation based on the achieved results.

First, we notice that no concept-containing unit, in these experiments a sentiment unit, emerges when the language model was trained on 0.2 million reviews. This is clear as the mean accuracy does not rise higher than 0.53. The model in general seems to have already learned to recognize sentiment in a very broad sense. This can be concluded as all scores are situated in the range between 0.6 and 0.7, which is better than simply guessing.

When trained on 2 million or more reviews, the respective language model seems

[5]After training these language models, we change the batch generation in the code due to discovering a bug. This means when retraining the language models, small changes in the results as well as in the needed time could be observed. Because of the time it would take to retrain all language models, we were not able to do this. However, we assume that this difference does not matter much in the overall picture.

to have evolved a sentiment unit with each of the chose hyperparameters. The scores of the classifier based on this unit alone often reaches the ones where all units are considered.

Second, we see that while there is a considerate improvement of all scores between 0.2 and 2 million reviews, we do not observe such a big leap between 2 and 20 million reviews. We may conclude that we experience a capacity ceiling with our implemented language model where the result does not get significantly better even if we use more training data. A factor for the lack of considerate improvement may be the ratio of positive to negative reviews as the 20 million review subset consists of 75% positive reviews.

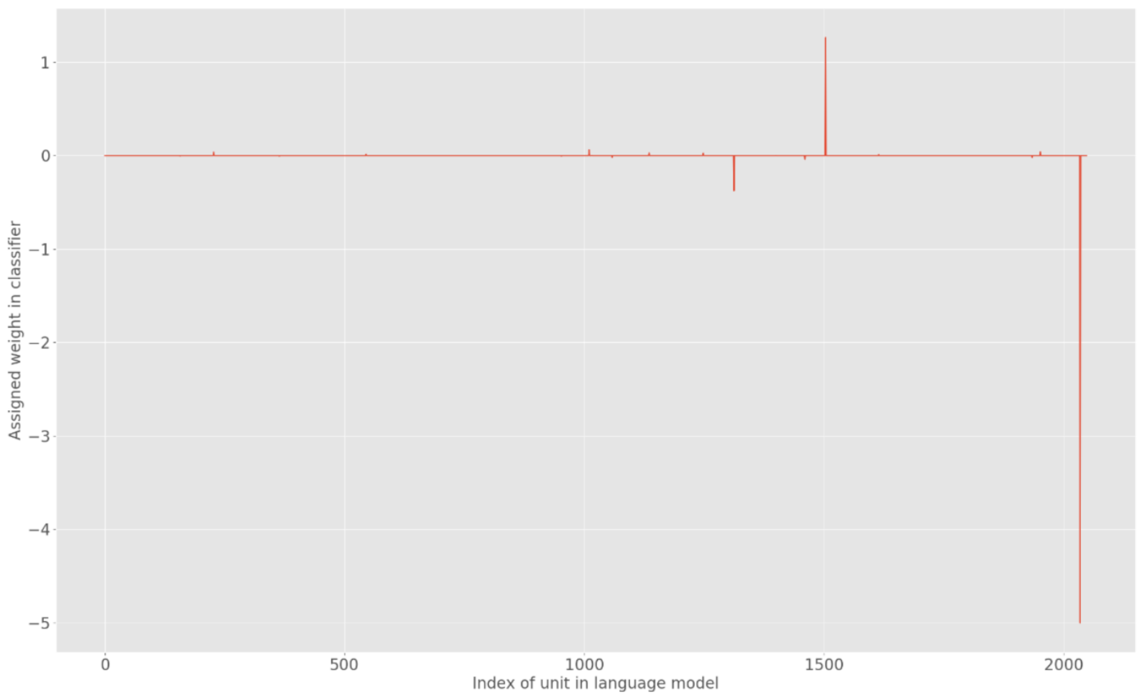
Third, we observe the importance of the chosen hyperparameters to be lopsided. The number of units in the hidden layer of the language model (*num_units*) seems to make quite a difference; increasing it almost always achieves better results, particularly raising it from 1024 to 2048. In contrast, the number of unrollings (*seq_length*) does not seem to have an obvious influence.

A visualization of the unit contributions of one of the best trained classifier can be seen in Figure 5.1a. We notice that the unit at index 2034 has by far the highest associated weight, i.e. this is our found sentiment unit. Another interesting detail is that of the 2048 possible features, the classifier only assigned 18 a non-zero weight. It seems that the language model distributed the analysis of sentiment mainly to one unit with only 17 supportive units. We may conclude that the higher the difference of the associated weight to the other weights, the better specialised the found unit is in analysing the examined concept. Subsequently, the result of a classifier using only this unit as feature should be good if the weight difference is high. In the conducted experiments where the language models were trained on 0.2 million reviews, we observe that the weight difference are overall small; this supports our theory. This can be seen exemplified in Figure 5.2a.

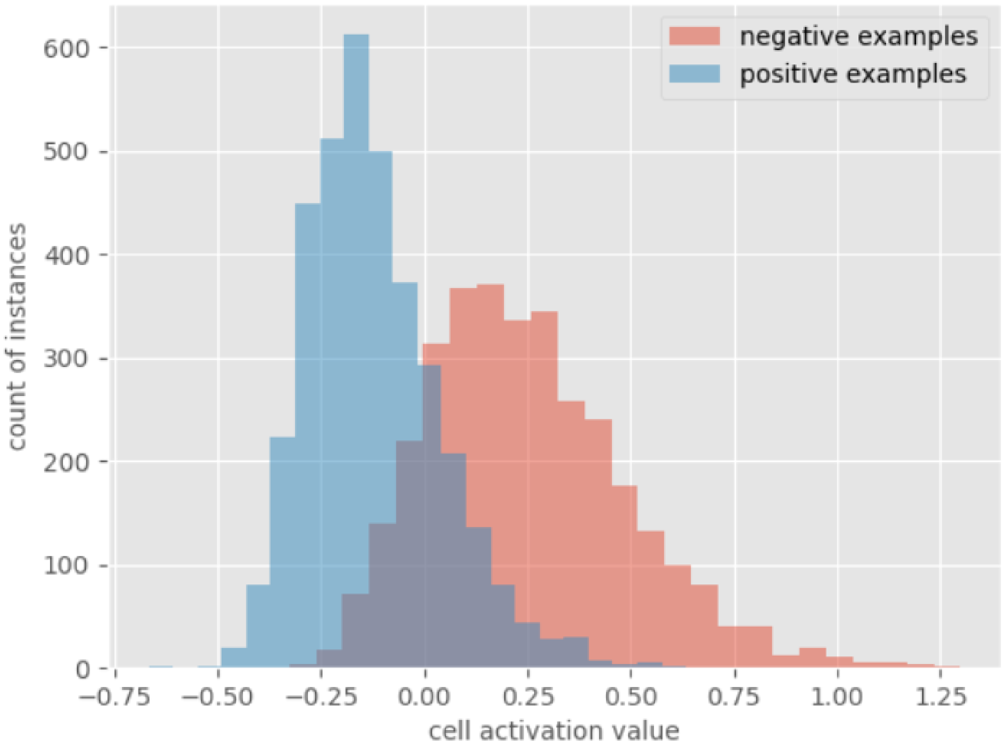
We visualized the distribution of the positive and negative instances which form the training split based on the cell activation value of this before-mentioned sentiment unit. This visualization can be seen in Figure 5.1b. We can see that even without a trained classifier the found sentiment neuron can distinguish between a positive and a negative phrase. An example how the according distribution of an undeveloped sentiment unit looks like can be seen in Figure 5.2b.

hyperparameters		all units used				only sentiment unit used				
seq_length	num_units	f1	recall	pred	acc	f1	recall	pred	acc	time
Language model trained on 0.2 million reviews										
100	1024	0.65	0.68	0.62	0.63	0.64	0.87	0.50	0.51	0.2h
	2048	0.67	0.71	0.66	0.65	0.64	0.82	0.52	0.53	0.7h
	4096	0.69	0.72	0.67	0.68	0.66	0.94	0.51	0.51	2.7h
200	1024	0.64	0.68	0.60	0.64	0.67	1.00	0.50	0.50	0.2h
	2048	0.64	0.69	0.59	0.61	0.67	1.00	0.50	0.50	0.6h
	4096	0.67	0.70	0.64	0.65	0.67	1.00	0.50	0.50	2.7h
Language model trained on 2 million reviews										
100	1024	0.75	0.76	0.73	0.74	0.71	0.80	0.65	0.68	2h
	2048	0.79	0.80	0.78	0.79	0.77	0.81	0.74	0.77	8h
	4096	0.83	0.84	0.82	0.83	0.81	0.84	0.79	0.81	32h
200	1024	0.71	0.74	0.69	0.70	0.66	0.65	0.66	0.66	3h
	2048	0.79	0.80	0.78	0.79	0.79	0.82	0.76	0.78	8h
	4096	0.82	0.83	0.80	0.81	0.79	0.83	0.75	0.78	33h
Language model trained on 20 million reviews										
100	1024	0.75	0.76	0.75	0.76	0.73	0.75	0.71	0.73	25h
	2048	0.85	0.88	0.81	0.84	0.84	0.89	0.89	0.84	79h
	4096	0.87	0.90	0.85	0.87	0.87	0.90	0.83	0.86	329h
200	1024	0.77	0.79	0.76	0.77	0.77	0.77	0.76	0.77	24h
	2048	0.85	0.86	0.84	0.84	0.79	0.83	0.76	0.78	80h
	4096	0.87	0.90	0.85	0.87	0.82	0.88	0.77	0.81	326h

Table 5.3: Performances of the logistic regression classifier on the binary SST dataset dependent on hyperparameters *seq_length* and *num_units*, and the amount of training data of the language model. In each case, the language model was trained for 1 epoch; the displayed time represents this duration. We compare the results of the classifier using the activation values of all units, with the results using only the activation value of the respectively found sentiment unit.

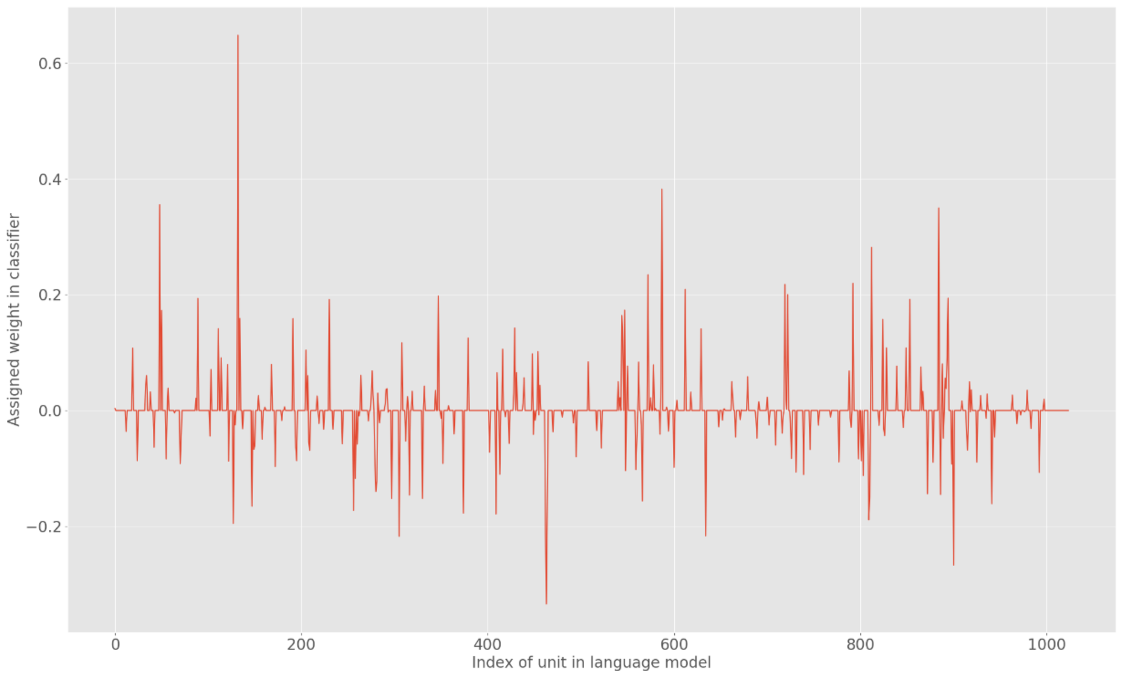


(a) Graph representing the unit contributions of the classifier.

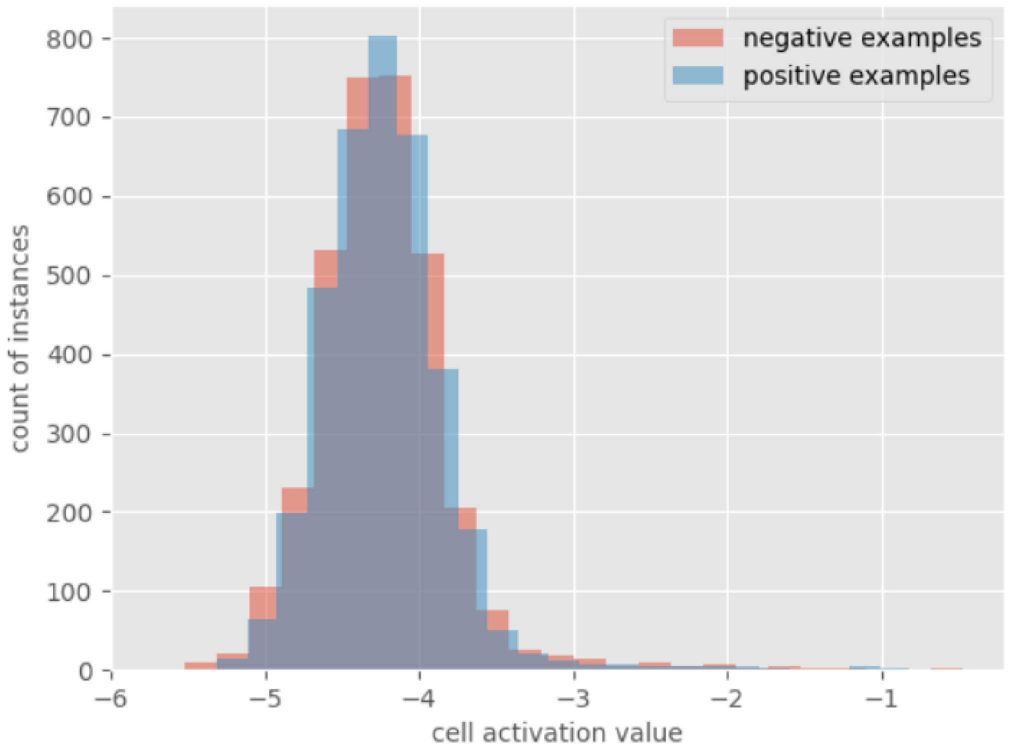


(b) Histogram representing the cell activation values for the found sentiment unit (index 2034) on the training split of the dataset.

Figure 5.1: Visualizations of a good classifier trained on the binary SST dataset. The associated language model was trained on 20 million reviews with *num_units* 2048 and *seq_length* 100.



(a) Graph representing the unit contributions of the classifier.



(b) Histogram representing the cell activation values for the undeveloped sentiment unit (index 132) on the training split of the dataset.

Figure 5.2: Visualizations of a mediocre classifier trained on the binary SST dataset. The associated language model was trained on 0.2 million reviews with *num_units* 1024 and *seq_length* 200.

As mentioned in section 3, we used another implementation of a language model for preliminary results while developing our own model. The results of these conducted experiments are in the range of those presented in Table 5.3, with small fluctuations on both sides. As mentioned before, these fluctuations may occur based on the random initialization of weights and biases in both implementations.

In our conducted experiments, we do not reach the result of [Radford et al., 2017] on the binary SST dataset, namely 91.8% mean accuracy using all units as features for the classifier and 90.2% using only the found sentiment unit. It is not clear if we would measure up to these results by also using all 80 million reviews of the Amazon product dataset; we would say its not very probable considering the lack of considerate improvement between 2 and 20 million reviews. Nevertheless, our results come near this number with the mean accuracy of the best classifier being 87% using all units and 86% using only the found sentiment unit. However, 87% on this particular dataset is below state of the art as many other approaches surpass it [Radford et al., 2017].

5.2 Spam Classification

Second, we will present our datasets and results on spam classification.

A challenge for our approach was finding matching datasets for our language model and our classifier. We did not find a large-scale dataset containing emails which were similarly formatted like the ones a smaller, labelled dataset. Therefore, we created our own datasets for the language model and the classifier using the pre-processed version of the Enron-Spam-dataset[6], introduced in [Metsis et al., 2006].

The training data for the language model consists of 23,220 emails; 8,175 spam and 15,045 non-spam, so called ham messages. The dataset for our classifier consists of 3,000 e-mails, equally consisting of spam and ham messages. It has to be noted that the length of these messages is very variable, partially because forwarded or previous messages are mostly included when the respective mail is a reply or the like. We create data split using 2100 examples for training, 300 for validation and 600 for testing, each with a 1:1 spam-ham-ratio. This means that guessing the same class for every instance of the testing set would amount to a mean accuracy of 50%.

[6]http://www2.aueb.gr/users/ion/data/enron-spam/

A short excerpt showcasing how these examples look like in their pre-processed form is presented in Table 5.4.

email	spam
Subject: young wifes click here to be removed	1
Subject: chart info here it is .	0
Subject: why pay for over priced pre \ scription dru @ gs ? ? ?	1
Subject: fw : revised michelle, sempra called on 21, 500 of needles space from 11 / 01 through 10 / 02 . please see attached memo from stepahie . thanks , tk [...]	0

Table 5.4: Excerpt from the pre-processed testing split of our created spam dataset.

We conduct our spam classification experiments on six language models. As before, those language models are trained with a different combinations of two hyperparameters respectively. We train all our models for five epochs respectively. The results of these experiments, following the process explained in section 4, are presented in Table 5.5.

In general, the results are very lopsided. All trained language models achieve very good scores when the activation values of all units are used. However, the accuracy of the respective developed spam unit, if it exists at all, varies to a large extend. The best found spam unit achieves a mean accuracy of 0.8 while the others lie within the range of 0.5 and 0.7. Regarding these spam units, we observe two things. First, language models trained with 200 unrollings achieve better results than their counterpart trained with 100 unrollings. Second, the scores get worse when increasing the number of units. These two findings are very surprising considering the experiments on sentiment analysis. There, the number of unrollings did not seem to have an obvious influence and the results actually got better with more units.

We have to keep in mind that the training data for the language model is relatively small, approximately 10 times smaller than the smallest training dataset for our sentiment experiments. This may be a possible explanation for our different results; the random initialization of the weight matrices and bias vectors may have a higher influence when there is not enough training data. Nevertheless, the large differences in performances of the found spam units surprise us.

hyperparameters		all units used				only spam unit used				
seq_length	num_units	f1	recall	pred	acc	f1	recall	pred	acc	time
100	1024	0.90	0.91	0.89	0.90	0.68	0.67	0.69	0.68	0.9h
	2048	0.93	0.96	0.91	0.93	0.63	0.71	0.56	0.58	2.1h
	4096	0.95	0.97	0.94	0.95	0.67	1.0	0.5	0.5	6.9h
200	1024	0.90	0.91	0.90	0.90	0.80	0.77	0.82	0.80	0.7h
	2048	0.93	0.95	0.91	0.93	0.66	0.70	0.63	0.64	1.9h
	4096	0.95	0.95	0.94	0.95	0.61	0.64	0.58	0.59	6.9h

Table 5.5: Performances of the logistic regression classifier on our spam dataset dependent on hyperparameters *seq_length* and *num_units*. In each case, the language model was trained for 5 epochs; the displayed time represents this duration. We compare the results of the classifier using the activation values of all units, with the results using only the activation value of the respectively found spam unit.

As with our sentiment classification experiments, we visualize the weights of the trained classifier and the distribution of positive and negative instances based on our found spam unit. For our best experiment on spam classification, these visualizations can be seen in Figure 5.3a and 5.3b respectively. We notice again an outstanding unit in the graph and a clear classification of positive and negative instances in the histogram.

We compare our results with those of the previously used implementation, as mentioned in section 3. With this implementation and used hyperparameters *num_units* 1024 and *seq_length* 100, we achieved a mean accuracy of 91% using all units and 87% using a found spam unit after training for five epochs. The result of this spam unit is a large improvements to all of the spam units found in our experiments, especially to the one with the same hyperparameters.

We see two possible explanations for this deviation. First, this good result could just be due to a very good starting constellation of the weights and biases. This would support our theory that the model has not been able to optimize the weights and biases because of the small amount of training data. Second, the other implementation could be actually better at evolving a concept-containing unit. One possible cause could be that our language model may be flawed when training over multiple epochs. This may be the case as this large discrepancy does not exist when training the language models for sentiment analysis for only one epoch. However, our language models still seem to grasp the concept of spam very well when using all units. It is very interesting to observe that a good representation using all units is, in some cases, caused by one main unit, while in other cases such a main unit does not seem to exist.

As we created our evaluation dataset ourselves, we have no direct comparison approach from other authors. This is why we implement a simple baseline algorithm. We use the bag-of-word-model approach and convert the examples of the labelled dataset to a vector consisting of tf-idf features[7]. Then, we train our linear classifier on these feature vectors. The documented results can be seen in Table 5.6.

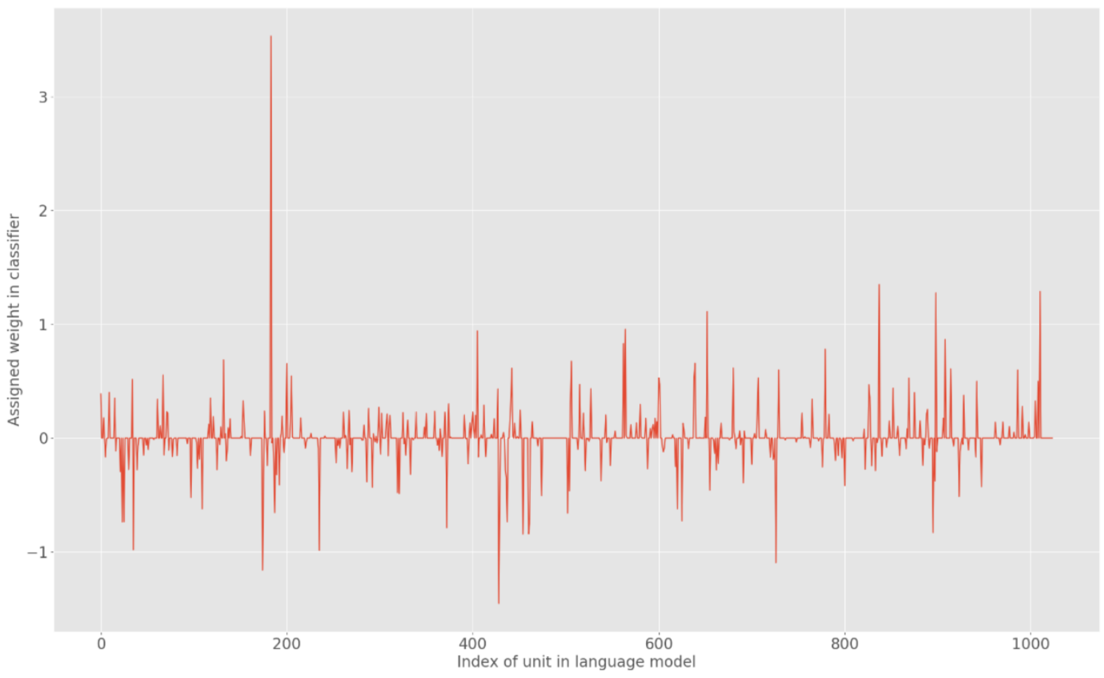
We can see that even a baseline algorithm achieves very good results on spam classification. In almost all our experiments, we reach and surpass most of those scores when training with the activation values of all units. Unfortunately, we observe that neither of our spam units approximates those scores.

However, we can still report the findings of evolved units in our language models which are responsible for analysing if a mail is spam or not.

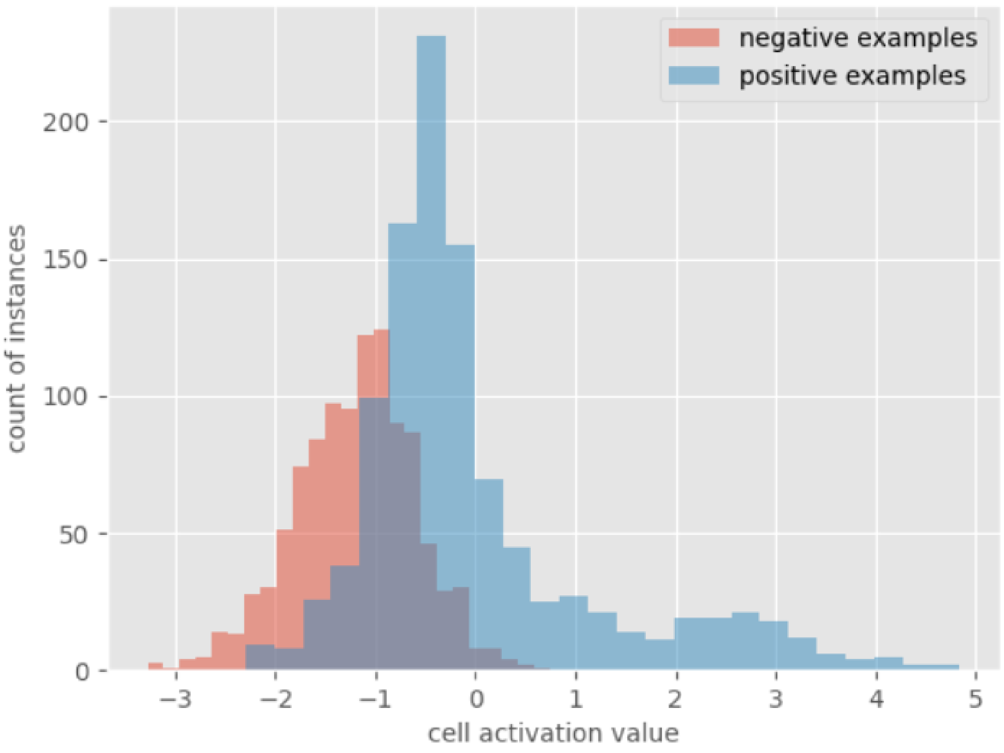
f1	recall	precision	accuracy	time
0.92	0.98	0.88	0.92	1s

Table 5.6: Performances of the baseline algorithm on our spam dataset.

[7]We used the submodule *sklearn.feature_extraction.text.TfidfVectorizer* in our implementation.



(a) Graph representing the unit contributions of the classifier.



(b) Histogram representing the cell activation values for the found spam unit (index 183) on the training split of the dataset.

Figure 5.3: Visualizations of a classifier trained on our created spam dataset. The associated language model was trained with *num_units* 1024 and *seq_length* 200.

5.3 Mood Classification

Last, we present our used datasets and achieved results on mood classification of music based on the lyrics analysis. In this specific case, we want to decide if a given lyric is happy (relation to positive themes) or sad (relation to dark and violent themes)[8] [Raschka, 2016].

We use the songdata dataset[9], containing 57,650 lyrics, for training our language model and the MusicMood dataset[10], introduced in [Raschka, 2016], for training and testing the classifier. Both datasets contain in their original format additional information like the artist and the song title. We parse these original file so that both used datasets consist of only the lyrics. In this case, we already pre-process the training data for the language model together with the dataset for our classifier as we change one detail: instead of replacing all newlines with whitespaces, we replace them with a "#". This was done to keep the structure of the lyric while still be able to pad the examples with newlines.

Originally, the MusicMood dataset consists of 1000 training and 200 testing examples. We split the training examples to achieve a train/validation/test split of 900 training, 100 validation and 200 testing examples. In contrast to the previously used evaluation datasets, this one is a little unbalanced; there are 400 positive and 500 negative examples in the training split. This has to be kept in mind when analysing the results. As the testing split consists of 105 positive and 95 negative examples, a classifier always predicting the positive class would achieve a mean accuracy of 52.5%. A short excerpt showcasing the pre-processed examples can be seen in Table 5.7.

As with the other two classification problems, we conduct our mood classification experiments with six language models trained on a combinations of two hyperparameters. The results of those experiments can be seen in Table 5.8.

[8]It has to be noted that these mood labels we are considering here are defined and manually assigned by the creator of the MusicMood dataset. It's questionable if those assigned labels are indisputable.
[9]https://www.kaggle.com/mousehead/songlyrics (18.06.2018)
[10]https://github.com/rasbt/musicmood (18.06.2018)

lyric	mood
Where, oh, where have you been, my love?#Where, oh, where can you be?#It's been so long since the moon has gone#And, oh, what a wreck you've made me## [...]	0
This kind of love makes me feel ten feet tall#It makes all my problems fall#And this kind of trust helps me to hold the line#I'll be there every time## [...]	1
I'm a pop star threat and I'm not dead yet#Got a super-dread-bet with an angel drug-head#Like a dead beat winner, I want to be a sinner#An idolized bang for the industry killer## [...]	0
Country day#A day in the unknown#A gentle breeze gently blowing#Country day#Country day#Another day in the unknown#I can feel it in my bones#Country day## [...]	1

Table 5.7: Excerpt from the pre-processed testing split of the MusicMood dataset.

It seems that none of our trained language models really grasps the concept of mood, barely (if at all) reaching the 0.6 mean accuracy mark when using the activation values of all units as features for the classifier. Considering this, it makes sense that the models do not seem to have evolved a mood unit. The closest one with such a mood unit might be the model trained with *num_units* 4096 and *seq_length* 100, which reaches a mean accuracy of 0.56. Fittingly, this model achieves the best results using all units, namely 0.64 mean accuracy. The according visualizations for this model can be seen in Figure 5.4.

It is also interesting to see that two of the undeveloped mood units predict the less probable negative class for all testing instances. Subsequently, the f1, recall and and precision scores are 0 and the mean accuracy is smaller than 0.5. This is most probably the case because the training data contains more negative than positive examples, as described above.

We see a possible cause in the overall underwhelming results in the amount of training data for the language model. It may be that with more lyrics to train on, the language model may evolve the ability to analyse the mood. However, it may also be the case that the mood of a given lyric is not an important enough feature crucial to truthfully predict the next character. In this case, more training data would not amount to a better result. Thus, the concept which the language model should learn to encapsulate may not have been a good choice.

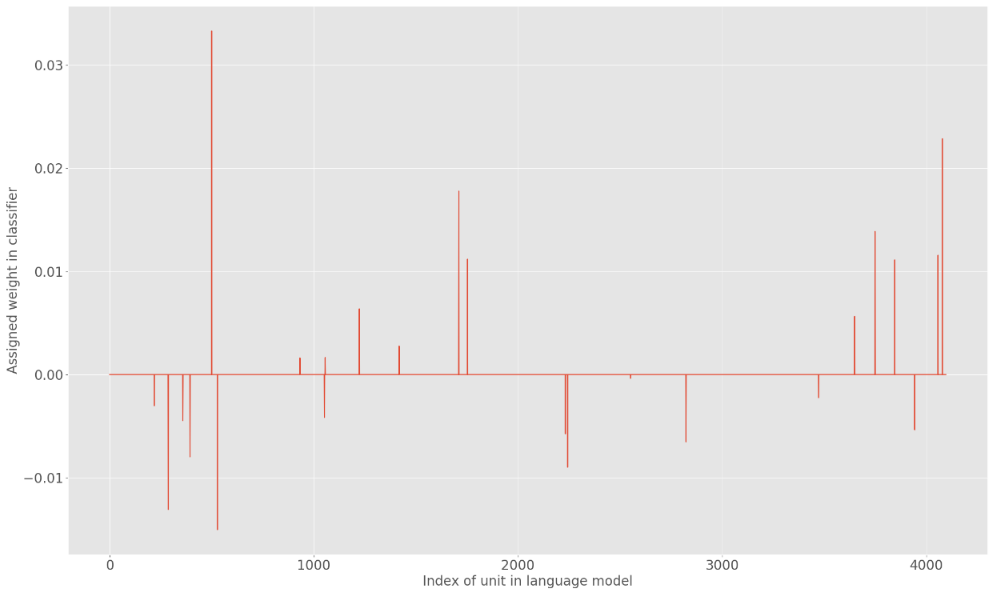
hyperparameters		all units used				only mood unit used				
seq_length	num_units	f1	recall	pred	acc	f1	recall	pred	acc	time
100	1024	0.32	0.21	0.69	0.54	0.33	0.22	0.64	0.53	2.2h
	2048	0.58	0.51	0.68	0.62	0.22	0.13	0.67	0.51	4.2h
	4096	0.60	0.50	0.73	0.64	0.40	0.29	0.68	0.56	12.7h
200	1024	0.43	0.32	0.64	0.55	0.07	0.04	0.67	0.49	1.6h
	2048	0.57	0.51	0.65	0.6	0.0	0.0	0.0	0.48	3.5h
	4096	0.60	0.56	0.63	0.60	0.0	0.0	0.0	0.48	12.1h

Table 5.8: Performances of the logistic regression classifier on the MusicMood dataset dependent on hyperparameters *seq_length* and *num_units*. In each case, the language model was trained for 5 epochs; the displayed time represents this duration. We compare the results of the classifier using the activation values of all units as features, with the results using only the activation value of the respectively found mood unit.

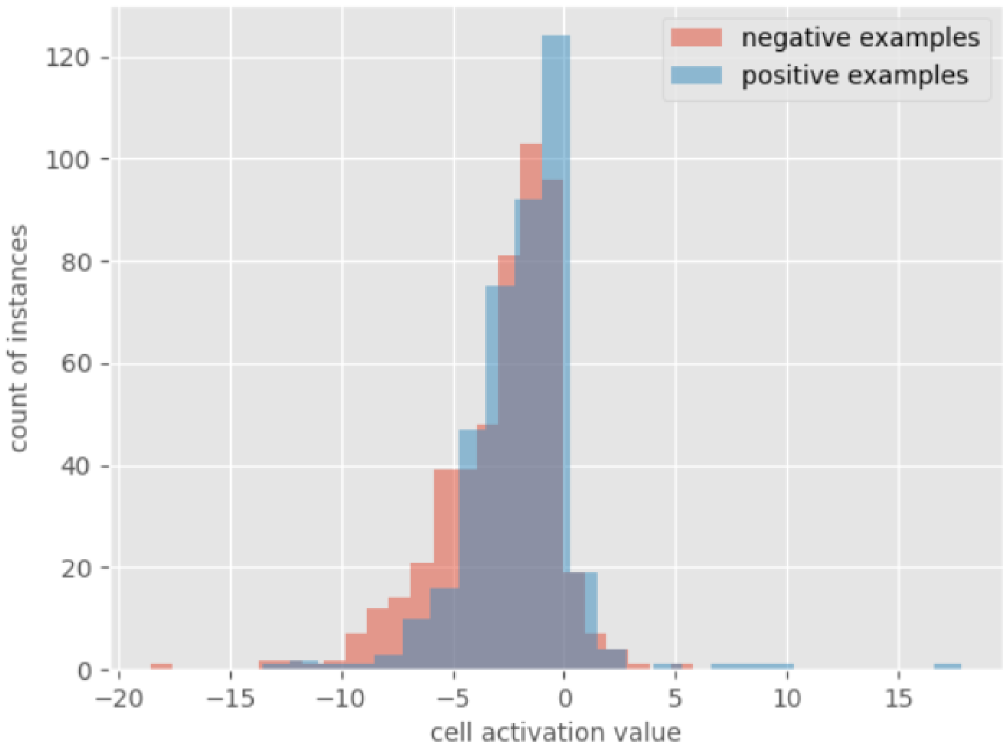
We compare our achieved results again with the preliminary results achieved with the previously used implementation. With hyperparameters *num_units* 1024 and *seq_length* 100, this implementation achieved 65% mean accuracy with all units and 61% with a found mood unit. As with spam classification, we notice that these results are better than the ones obtained with our own implementation. This supports our theory that our language models is flawed when training over multiple epochs.

In our conducted experiments, we do not reach the reported result of [Raschka, 2016], which is a 72.5% mean accuracy on the testing set[11]. As described before, we may approximate this value when training our language model on a larger amount of lyrics.

[11] The results described in [Raschka, 2016] are different from the ones reported on the corresponding GitHub page https://github.com/rasbt/musicmood (22.06.2018). As the results on the GitHub page seem more recent, we consider those.



(a) Graph representing the unit contributions of the classifier.



(b) Histogram representing the cell activation values for the found mood unit (index 500) on the training split of the dataset.

Figure 5.4: Visualizations of a classifier trained on the MusicMood dataset. The associated language model was trained with *num_units* 4096 and *seq_length* 100.

5.4 Analysis of complexity

In this section, we will discuss the basic complexity of our language model and classifier.

The computational complexity of training a LSTM network is depicted as $O(B^2 * S^2)$ where B is the number of LSTM memory cell blocks in the hidden network and S the number of LSTM memory cells in those blocks [Gers, 2001]. This holds when keeping the number of units in the input and output layer fixed. As described in section 3, we use *num_units* memory cell blocks of size 1. Therefore, the complexity of our language model is $O(num\_units^2)$. This can also be observed when looking at the running time of the language models depicted in Tables 5.3, 5.5 and 5.8. When doubling *num_units*, we notice a roughly four times increased runtime. This makes sense as $O((num\_units * 2)^2) = O(4 * num\_units^2)$.

Regarding the space complexity, [Gers, 2001] also states the overall number of weights as $O(B^2 * S^2)$, assuming the number of units in the output and input layer stay fixed. Subsequently, this amounts again to $O(num\_units^2)$ in our implementation. This correlates with the size of the language model savefiles which contain the respective weights of the networks. Independent of the training data and other hyperparameters, the savefiles of the language models have approximately a size of 20 MB if *num_units* is 1024, 70 MB if *num_units* is 2048 and 260 MB if *num_units* is 4096. We notice again the quadrupling size.

When considering the space complexity of our implemented language model, we also need to note the fact that, after initially reading the whole training data, it remains in the memory for the whole duration of the training process. As the according training file can potentially be multiple gigabytes in size, we consider the size s of the file as an offset to the previous complexity of $O(num\_units^2)$. This results in a maximal space complexity of $O(num\_units^2 + s)$, assuming the units in the output and input layer stay fixed.

Regarding the runtime of our implemented classifier, we observe the most time-consuming part to be the extraction of the cell states which form the feature vectors for the respective data splits. The needed time depends on the number of units, the number of examples in the respective split and the length of these examples. An overview of the runtime of the classifier trained for sentiment classification can be seen in Table 5.9. The classifiers were trained on a Personal Computer with an

Intel(R)Core(TM)i7-6700HQ CPU @ 2.60GHz processor and 16GB RAM. We can see that the runtime is linear regarding the number of examples in the respective split. When doubling the number of hidden units from 2048 to 4096, we notice a big leap in the needed time. The runtime seem to quadruple, like we have seen when training the language model, but we do not see this between 1024 and 2048 units. There, the runtime only doubles.

	time spend on feature vector creation			
num_units	training split	validation split	testing split	total
1024	20min	4min	7 min	33 min
2048	50min	8min	16min	75 min
4096	235min	40min	81min	360min

Table 5.9: Average runtime of the sentiment classifier. Together with the total running time, we document the time spend on the creation of the feature vectors for the respective splits. In the used binary SST dataset, the training split consists of 6920, the validation split of 872 and the testing split of 1821 examples.

6 Conclusion

We have implemented a system, consisting of a neural language model and a linear classifier, which can find potential semantic units in the trained language model. The evaluation has shown that the system is able to find units in relation with different concepts, if such units have evolved during training.

On sentiment analysis, our trained language models were able to produce close results to those of [Radford et al., 2017], evolving their own sentiment unit responsible for these good results. On other text classification tasks, our results are lopsided. While our trained language models achieve very good results on spam classification when using all units, the performance of their evolved semantic units differ to a great extent. On mood classification, our trained models do not seem to evolve such semantic units.

In the following, we will provide suggestions which could improve our overall results.

- **Train on more data.** As we have seen when evaluating the language models on sentiment analysis, more training data equals a better trained language model. We assume that the results on all three classification problems can be improved when at least doubling the training data. This will take a lot of time, at least for sentiment classification, considering that our best model in the sentiment experiments needed approximately 14 days to train on 20 million reviews. With the documented time, we estimate that the training time of a model with the same hyperparameters quadruples when training on all 83 million reviews of the Amazon product dataset.

- **Change the feature vector.** In the current implementation, we use the cell state of the language after processing the given text as the feature vector. Instead, we could save intermediate values while processing the given text and build the feature vector based on these multiple values. We estimate that this will not take a long time to implement. In fact, we have started with this implementation but could not finish it in time. When testing a preliminary version, we got worse results with multiple values than with only one value.

- **Revise the implementation of the language model.** It seems that our language model does not work as well as other implementations when training over multiple epochs. A next step to improve the results on mood and spam classification would be to look over the code and see if it can be optimised. We estimate that this suggestion would take a few days.

List of Figures

List of Tables

Bibliography

[Bengio, 2008] Bengio, Y. (2008). Neural net language models. *Scholarpedia*, 3(1):3881. revision #91566.

[Bengio et al., 2013] Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828.

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3(Feb):1137–1155.

[Cambria et al., 2013] Cambria, E., Schuller, B., Xia, Y., and Havasi, C. (2013). New avenues in opinion mining and sentiment analysis. *IEEE Intelligent Systems*, 28(2):15–21.

[Gers, 2001] Gers, F. A. (2001). Long short-term memory in recurrent neural networks. *Unpublished PhD dissertation, Ecole Polytechnique Fédérale de Lausanne.*

[Gers et al., 1999] Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with lstm. *9th International Conference on Artificial Neural Networks.*

[He and McAuley, 2016] He, R. and McAuley, J. (2016). Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *Proceedings of the 25th International Conference on World Wide Web*, pages 507–517.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

[Kiros et al., 2015] Kiros, R., Zhu, Y., Salakhutdinov, R. R., Zemel, R., Urtasun, R., Torralba, A., and Fidler, S. (2015). Skip-thought vectors. In *Advances in Neural Information Processing Systems*, pages 3294–3302.

[Le and Mikolov, 2014] Le, Q. and Mikolov, T. (2014). Distributed representations of sentences and documents. In *International Conference on Machine Learning*, pages 1188–1196.

[Lubitz, 2017] Lubitz, M. (2017). Who drives the market? Sentiment analysis of financial news posted on Reddit and Financial Time. *Bachelor's Thesis, Albert-Ludwigs-University, Department of Computer Science.*

[Metsis et al., 2006] Metsis, V., Androutsopoulos, I., and Paliouras, G. (2006). Spam filtering with naive bayes - which naive bayes? In *Third Conference on Email and Anti-Spam.*

[Mikolov et al., 2013] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv:1301.3781.*

[Mikolov et al., 2010] Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association.*

[Radford et al., 2017] Radford, A., Jozefowicz, R., and Sutskever, I. (2017). Learning to generate reviews and discovering sentiment. *arXiv:1704.01444.*

[Raschka, 2016] Raschka, S. (2016). MusicMood: Predicting the mood of music from song lyrics using machine learning. *arXiv:1611.00138.*

[Schütze et al., 2008] Schütze, H., Manning, C. D., and Raghavan, P. (2008). *Introduction to Information Retrieval.* Cambridge University Press.

[Socher et al., 2013] Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., and Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642.

[Yuan et al., 2012] Yuan, G.-X., Ho, C.-H., and Lin, C.-J. (2012). Recent advances of large-scale linear classification. *Proceedings of the IEEE*, 100(9):2584–2603.