Semantic Text Matching Using Convolutional Neural Networks

Runfen Wang

Uppsala University
Department of Linguistics and Philology
Master’s Programme in Language Technology
Master’s Thesis in Language Technology
October 1, 2018

Supervisors:
Aaron Smith, Uppsala University
Alexander Solsmed, Lunchback
Abstract

Semantic text matching is a fundamental task for many applications in Natural Language Processing (NLP). Traditional methods using term frequency-inverse document frequency (TF-IDF) to match exact words in documents have one strong drawback which is TF-IDF is unable to capture semantic relations between closely-related words which will lead to a disappointing matching result. Neural networks have recently been used for various applications in NLP, and achieved state-of-the-art performances on many tasks. Recurrent Neural Networks (RNN) have been tested on text classification and text matching, but it did not gain any remarkable results, which is due to RNNs working more effectively on texts with a short length, but long documents. In this paper, Convolutional Neural Networks (CNN) will be applied to match texts in a semantic aspect. It uses word embedding representations of two texts as inputs to the CNN construction to extract the semantic features between the two texts and give a score as the output of how certain the CNN model is that they match. The results show that after some tuning of the parameters the CNN model could produce accuracy, prediction, recall and F1-scores all over 80%. This is a great improvement over the previous TF-IDF results and further improvements could be made by using dynamic word vectors, better pre-processing of the data, generate larger and more feature rich data sets and further tuning of the parameters.
# Contents

Acknowledgements .................................................. 4

1 Introduction ......................................................... 5
   1.1 Background .................................................. 5
   1.2 Outline ..................................................... 6

2 Related works ....................................................... 7

3 Main concepts and theory ......................................... 8
   3.1 TF-IDF ....................................................... 8
   3.2 Word embedding and word2vec ................................ 8
   3.3 CNN ......................................................... 9

4 Experiments .......................................................... 11
   4.1 Data collection ................................................. 11
   4.2 Pre-processing ................................................. 11
      4.2.1 Expanded data .......................................... 11
      4.2.2 Word2vec ............................................... 12
   4.3 CNN model .................................................... 12
   4.4 Parameter settings ............................................ 13
      4.4.1 Default settings ....................................... 13
      4.4.2 Experiments on multiple settings ....................... 14
      4.4.3 Best combination of parameters ........................ 15

5 Results and discussion ............................................. 16
   5.1 Results for different parameter settings ..................... 16
      5.1.1 Data content ............................................ 16
      5.1.2 Positive and negative pair ratios ....................... 17
      5.1.3 Training and testing data ratios ......................... 19
      5.1.4 Filter size .............................................. 20
      5.1.5 Number of filters ...................................... 21
      5.1.6 Number of neurons ..................................... 22
      5.1.7 Dropout ................................................. 23
      5.1.8 Batch size .............................................. 24
      5.1.9 Number of steps ........................................ 25
      5.1.10 Learning rate ......................................... 26
   5.2 Final results .................................................. 27

6 Conclusion and future work ....................................... 29
Acknowledgements

I am truly grateful that my supervisor at Uppsala University was Aaron Smith, not only because he guided me with brilliant suggestions and valuable materials, but also because of his professionalism and positive attitude which was very important to me. In addition to all encouragement and support, he also gave me enough space to plan and seek my ideas. Working with him was a pleasant journey.

I would also like to thank my supervisor Alexander Solsmed at Lunckback, for giving me such an opportunity to work with them, and for providing me with a great thesis topic and all data I needed.

Lastly, thanks to my partner Joel for all spiritual encouragements throughout the whole thesis process, and who at the last moment helped me resolve some issues in Latex.
1 Introduction

1.1 Background

This paper is a master thesis for Uppsala University, performed at Lunchback in Stockholm, Sweden. Lunchback is a web service to help users expand their professional network by arranging physical lunch meetings between relevant startups and investors. Therefore, to provide a more precise match for users is one of the main tasks and challenges for the service. Two primary techniques are applied to the issues of the matching. One is simply checking through user’s information including profile and website’s descriptions, and manually label the most relevant ones as matched pairs. Another one is to match keywords by using TF-IDF implementation based on user’s data.

The problems of these two methods lie in the facts that manually matching can be a time-consuming process and less objective, especially when the data is large and it is automatically getting bigger when new users keep signing into the service. For TF-IDF implementation, it benefits matching exact keywords between texts, and ignores semantic similarities in documents which plays a significant role in text matching. In Lunchback’s service, startups tend to describe their information with explicit and specific vocabulary, and investors usually use more implicit and vague words to express what they are looking for, it creates difficulties for the precise matches.

To address the issues expounded above on the former methods, I decide to apply a CNN model to complete text matching for this web service. CNN systems have proved to have great performances on various applications in NLP, e.g. classification and image recognition, even on text matching as image recognition. Before I start building a CNN structure, I need to convert manually labeled data to word embedding vectors. Each user’s data based on their own information will be converted to a matrix, and later put into the CNN as input 1, their corresponding match will be converted to another matrix and put into the system as input 2. The matrices for each pair are passed into the system and traverse through the different layers. In the end, the CNN system will decide if input 1 and input 2 matches. The output will be classified to two classes which are represented with numeric values whether it is a match [0,1] or a non-match [1,0]. The sum of the two indexes is 1, meaning they express the probability for a match versus a non-match. The class with the highest probability will be selected as the outcome for the pair. A simplified model structure can be seen in section 4.3.

In this thesis, the following research questions will be addressed:

1. Can a CNN model be used for semantic text matching?
2. How can a CNN model be improved and better results be gained?
3. Can a CNN model outperform TF-IDF model in the task of matching user pairs?

To answers these questions, first a CNN structure with default settings has to be built to match users, then multiple tasks have to be performed to improve the matching results, including varying data sets and tuning multiple parameters within the CNN structure. Finally, a measurement metric combined with figures showing comparisons between the CNN model and TF-IDF model.

1.2 Outline

This thesis covers 6 main sections and is structured as follows:

• Chapter 1 is an introduction that gives a basic background and aim of the thesis, at the same time it points out the issues the previous methods used in the service and proposes a novel solution which is a CNN structure. Furthermore, it briefly states the pre-process and main process in the system.

• Chapter 2 is a brief review of related work on CNN within NLP.

• Chapter 3 focuses on the main theoretical notions and mathematic equations of the model used for the main task in the thesis, meanwhile it covers various concepts, namely word embedding (WE), TF-IDF. Furthermore, this chapter also outlines plain theories and methods within Neural Networks (NNs) and CNNs.

• Chapter 4 describes data collection, word embedding as pre-processing, and default settings first, then running experiments on different settings including data selection, word embedding choice, and tuning of multiple parameters within the CNN model.

• Chapter 5 lays out results related to multiple experiments on different settings, then combining all the best settings for system and obtaining a final result, comparing with the results based on default settings and the baseline based on TF-IDF model provided by Lunchback. Furthermore, a discussion of the effects of all results.

• Chapter 6 describes possible improvements for the future work and suggests possible ways to expand the research further.
2 Related works

CNNs have recently been applied to various applications within NLP and achieved impressive results. The most applications of CNNs are related to classification tasks, such as sentence or text categorization, sentiment analysis or spam detection.

This thesis is most similar to the previous work of CNNs for sentence classification (Kim, 2014), where they train a simple CNN with small hyperparameter tuning on pre-trained word vectors trained with word2vec models for sentence-level classification tasks, and gain state-of-the-art results on multiple benchmarks. The paper also explore experiments with both static and dynamic word embedding. A similar, but more complex Dynamic Convolutional Neural Network (DCNN) is used to capture word relations of varying size, and achieves high performance on sentiment classification (Kalchbrenner et al., 2014). CNNs are presented jointly with character-level, word-level and sentence-level representations to also perform sentiment analysis, but on short texts (Dos Santos and Zadrozny, 2014).

For text categorization, CNNs are not trained on pre-trained word vectors but instead on one-hot vectors (Johnson and T. Zhang, 2014).

How to choose hyperparameters in CNN architecture is a big challenge for researchers, such as the choices of input representation, or number and sizes of filters, activation functions, pooling strategies, and so on. Author Zhang and Wallace provide a practical framework of CNN architecture for sentence classification, the results are based on extensive experimental analysis on the effects of varying hyperparameters. Some results are very interesting and give a great inspiration to my thesis, e.g. max pooling is always better than average pooling, and regularization does not have big impact on the results (Y. Zhang and Wallace, 2015). CNNs have also been used for relation extraction and relation classification tasks (Nguyen and Grishman, 2015) (Sun et al., 2015) (Zeng et al., 2014).

For information retrieval tasks, CNNs are trained to learn semantically meaningful representations of sentences (Gao et al., 2014) (Shen et al., 2014).

For part-of-speech tagging tasks, CNNs are used to extract character-level features and join with word-level embeddings, and obtains a state-of-the-art POS tagger for many languages without any handcrafted features (Dos Santos and Zadrozny, 2014). Even without cooperation with word-level embedding, CNNs can learn directly from character-level embedding and apply it for sentiment analysis and text classification tasks, and gain competitive results for large datasets compared to traditional models such as bag of words, TF-IDF, and even deep learning models such as word based CNNs (X. Zhang et al., 2015).

Finally, CNNs are utilized on text matching tasks where the text matching is treated as image recognition within CNN. Authors develop a new deep architecture named MatchPyramid to automatically capture matching patterns and experimental results show that their model outperformed some other deep learning algorithms (Pang et al., 2016).
3 Main concepts and theory

3.1 TF-IDF

In information retrieval it is often important to rank documents after their relevance regarding a search term. A simple method for doing this is the Term Frequency (TF), i.e. how often the search term appears in each document, and ranking the documents after the total score for all search terms. TF can be defined in different ways, such as the total number of occurrences of the term in a document or the frequency of the term in a document, i.e. the ratio of occurrences of the term and total occurrences of all terms. A drawback to this method is that certain search terms can be more informative than others, where the frequency of a word such as "the" may not be indicative of the relevancy of a certain document for a search term such as "the Senate".

JONES (1972) introduced the concept of Inverse Document Frequency (IDF) as a weighting scheme to adjust for this bias for common terms. IDF means that when scoring a document for a certain search term the score will additionally be weighted by the logarithm of the ratio between total number of documents and documents containing the term. A term that occurs in every document will therefore receive a weight of zero, where the reasoning is that if it occurs in every document it does not provide much information of the relevancy of any single document. If on the other hand the term occurs in very few documents the weight will grow logarithmically with the ratio and increase the importance of rare terms in the scoring of a document’s relevance. The TF-IDF for a given document d in a collection of documents D, where the total amount of documents is N, with the search term t is:

\[
TFIDF(t, d, D) = TF(t, d) \cdot IDF(t, D) = f_{t,d} \cdot \log \left( \frac{N}{\{d \in D: t \in d\}} \right)
\] (1)

To handle the case where the term does not occur in any documents the denominator can be redefined as \(1 + \{d \in D: t \in d\}\).

3.2 Word embedding and word2vec

Word embedding is a collection of methods in Natural Language Processing for making vector representations of words where the idea is to map words into a vector space where similar words get grouped together.

One of the simplest methods for word embedding is the one-hot embedding scheme wherein each word is represented with a vector of the same length as the total number of unique words in the corpus. The vector is then filled with zeros except for one position which corresponds to the position of the word in an ordered list of all unique words. The zero at this position is changed to one, hence
the name "one-hot". This results in a very sparse vector with possibly thousands of zeros for a decent size corpus.

In Mikolov et al. (2013) from Google created a word embedding model based on the continuous skip-gram model but with several extensions. This model became word2vec which is a popular word embedding model in modern NLP applications. It is trained using logistic regression in which a neural network model is taught to pick the context for a given word from some generated noise. This training data and noise is generated from a corpus and the training is unsupervised, meaning the data does not need to be labeled in any way before the training. The vectors generated from the word2vec model are both much smaller and denser than those typically created using one-hot embedding on the same corpus. They also tend to capture a surprising amount of semantic information of the words. For example, Mikolov et al. showed that their model managed to group countries with their capitals without providing any supervised information of what a capital or country is.

3.3 CNN

Convolutional Neural Networks are an extension of the ordinary Neural Networks inspired by the vision processing in living organisms. The idea behind CNNs were developed over decades by many different contributors, most notably (LeCun et al., 1989) and (Lecun et al., 1998). A CNN generally has the following structure: an input layer, a convolutional layer, a pooling layer, a fully connected (also called dense) layer and finally an output layer. The benefits of CNNs are the spatial invariance and automatic feature generation that comes from the use of learned filters for the convolution step and the use of pooling. This means that CNNs automatically generates features in the learning process and that these features are spatially invariant, for example a straight line in a picture will be recognized even if it is moved. For the same reason CNNs are best used for input data that has some ordering, such as pixels in a picture or words in a sentence. Data which can be shuffled without losing any information will not benefit from the convolution and pooling steps.

The convolution layer consists of filters that will be convoluted with the input to generate feature maps. A filter is a matrix with predetermined dimensions and filled with numbers that are initialized randomly and later learned through the training. During the convolution the filter is swept with a given stride length over the input and for each step a convolution is made which results in a number that is put into the feature map.

In the pooling layer a pooling window with a predetermined size and stride length is swept over the feature map. For each step the numbers inside the pooling window are condensed into a smaller set of numbers depending on the pooling strategy. The most common strategy is max-pooling where only the largest number in the pooling window is kept. The pooling step concentrates the information in the feature map and makes it less dependent on the position of the features in the original input.

The convolution and pooling steps can be repeated several times but finally you will end up with a set of feature maps which are then flattened into a single
vector. This vector is given as input to the dense layer, which is an ordinary Neural Network that consists of one to many hidden layers. Each number in the input vector is passed to every neuron in the hidden layer. Every connection has an associated weight which is multiplied to the number. When all numbers have been passed to a neuron they are summed together with a bias added. They are then passed through an activation function, most commonly ReLu (rectified linear units), to capture any nonlinear relations. Here an optional dropout step can be added where every neuron has a certain chance to drop its value, where the idea is that this might help against overfitting to the training data. Every neuron in the hidden layer is finally connected to every neuron in the output layer where the final result can be determined by seeing which neuron is most activated by the input.

Training of the CNN is done by comparing the output of the model to the label of the input. A loss function determines the discrepancy between the output and the label and a loss minimization method, often different variations of backpropagation, adjusts the weights, biases and filters to minimize this loss.
4 Experiments

In order to answer the research questions, the following experiments are outlined. To be more explicit, section 4.3 answered question 1, and section 4.4 answered question 2, lastly question 3 is explored in section 5.

4.1 Data collection

The main data sets used in the project are collected from three events which were organized by Lunchback in Stockholm during 2017. Users’ information include profession, a summary that briefly describes what the submitter is good at and looking for, as well as a short answer to the query that Lunchback requires users to fill in. In addition to all the users’ information collected above, Lunchback also checked through users’ other possible information from other social media, such as blogs, LinkedIn, Facebook and so on, then manually matched the most relevant ones to the matched pairs. Three data sets are named Event1 which consists of 348 matched pairs, Event2 contains 38 matched pairs, and Event3 has 69 matched pairs.

4.2 Pre-processing

4.2.1 Expanded data

For CNN models to learn as many features as possible from the data sets, and to determine if the input users are matched or not afterwards, the model needs to learn from negative matched pairs as well, therefore creating negative matched pairs are necessary. Based on the three existing positive data sets and assuming all other pairs of users are non-match pairs, we can obtain a great amount of negative matched pairs. To balance the learning features for the model, I decided to only randomly select the same number of negative pairs as the positive pairs in the data set. Hence, up to this step, Event1 has increased to 696 pairs with 50% positive pairs and 50% negative pairs shuffled randomly together, Event2 and Event3 have gathered 76 and 138 shuffled negative and positive pairs respectively. Due to the small size of the data sets and the need of the later experiments, I combined all three data sets to one big data named Combined Data that covers 910 pairs in total.

Due to the total number of the negative pairs being over two thousand and the training process only needs 455 pairs out of them, the system is made to automatically select 455 pairs randomly for each training session. During the processing, the system will first shuffle the selected negative pairs with the 455 positive pairs and then divide the whole data set into two parts based on the training requirements, whether 90% for training and 10% for testing as the default
setting or 50% for each data set. In this way what pairs end up in which data set is random and may be different every time. This is process is done for all experiments, including the final one with the best settings.

### 4.2.2 Word2vec

For the word2vec embedding Google’s pre-trained model available at the word2vec website was used. The model is trained on roughly 100 billion words from a Google News dataset and contains 300-dimensional vectors for 3 million words and phrases. This model remained static during training of the CNN, meaning the word2vec-vectors were not updated as part of the training process. The word2vec-vectors for a user are combined into a matrix which is used as input for the CNN.

### 4.3 CNN model

Since the input of the CNN model is the matched pairs, one pair is taken by the system at a time, and thus the model is constructed with two parallel layers, see Figure 1. Layer1 takes user1 as input1, meanwhile layer2 will take user2 as input2. Both inputs will then be passed to the convolution layer1 and convolution layer2, which generates the feature maps for user1 and feature maps for user2. All feature maps from both convolution layers will pass through the pooling layer where one-max-pooling is used as the pooling method. One-max-pooling is a method that removes all numbers in a feature map and only the largest number is kept. After all remaining numbers are collected from the pooling layer they will be flattened into one big list, and passed as input into the fully connected layer, also called the dense layer. The dense layer consists of only one hidden layer in this project which is directly connected to the two neurons in the output layer. ReLu activation is used as well as dropout. The first neuron in the output layer corresponds to a negative match and the second to a positive match. The label for a match is therefore [0, 1] while a negative match is [1, 0].
4.4 Parameter settings

In this section, first a default setting will be defined to be used as a basic setting from which the effect of varying one parameter at a time can be studied. The point of altering only one parameter for each experiment is to see how the current parameter affect the performance and try to seek out the best settings for the model. Secondly, the set of parameters which will be investigated will be described.

4.4.1 Default settings

Default settings consist of three parts which are data set settings, word embedding settings and CNN model settings.

For the data set settings, as described in section 4.1 and 4.2, Combined Data is default and consists of profession, answer and summary from the original combined user data. It contains 50% negative pairs and 50% positive pairs. Before running the model 90% of the data is randomly selected as the training data, and the rest of the data is withheld as testing data.

In the pre-processing phase, I chose Google’s pretrained word2vec vectors for word embedding.

For the CNN model settings, parameters are determined for the different layers and for training strategies. In the convolution layer, two parameters are determined which are the filters size and the number of the filters. The filters size are 3, 5, 7, and each size will have 32 filters. In the pooling layer, one-max-pooling is the default. In the dense layer, 1024 neurons are default in the hidden layer. ReLu activation is used and a dropout rate of 0.4. For training the input data is packed into batches of 50 with 1000 steps and a learning rate of 0.01.
4.4.2 Experiments on multiple settings

For every parameter setting that was tried the model was run five times and the mean scores and their standard errors were calculated and are presented in section 5.

Data content

In order to understand which information are most important for the training data, and to improve any future work for data selection for the model, it is trained on a variation of the user data which consists of Profession, Answer and Summary. Five different combinations in total were tested in addition to the default: only Profession; only Answer; only Summary; Profession and Answer; Profession and Summary; Answer and Summary.

Positive and negative pair ratios

The negative matches generated in the pre-processing of the data far outnumbered the positive matches. For the default settings a ratio of 1:1 between positive and negative matches were chosen, but here I want to see how different ratios affect the scores. I start with varying the ratio for the training data while the testing data remains at a 1:1 ratio, however in a real-world application of the model every pairing of user must be investigated. Therefore, I repeat the experiment but save all remaining negative pairs and add them to the testing data to see how the training ratio affects the performance on this more realistic testing data where negative pairs far outnumber the positives ones.

Training and testing data ratios

In addition to the default split of the data to 90% for training and 10% for testing I will here investigate the effect of different splits. I want to find the balance between having enough training data to get good scores but enough testing data to get reliable scores.

Filter size

The filter size basically represents the n-gram size of the features you try to capture. As the default value I selected a range of sizes trying to capture both short and long distance relations within the sentences. In this experiment I will try and see if other filter sizes work better for this type of data.

Number of filters

The number of filters determine how many different features you can find of the same n-gram type. A lower number of filters will speed up the training process but you risk missing some potential features. On the other hand there should be a tipping point where extra filters will only be redundant and at best slow down the training and at worst dilute the more important features. I will therefore try to both lower and increase the number of filters.
Number of neurons

The benefit of a high amount of neurons in the hidden layer seems to be very different depending on the application if you study the literature on neural networks. Similarly to the number of filters there should be a minimum and a maximum amount of neurons below and above which will negatively affect the score. The number of neurons will also affect the training speed. I will therefore try to both increase and decrease the number of neurons compared to the default value.

Dropout

I will here see if the ratio of the dropout affects the scores and whether there are any benefits to dropout at all for this model.

Batch size

Batch size determines how many pairs will be run through the model until the weights, biases and filters are updated. A large batch size means more pairs will be used when making a change to the model which should lead to a more even training process. The batch size strongly affects the training time however and I will try and find the balance in this experiment.

Number of steps

The number of steps is a balance between successful training and overfitting. It also affects the training time. I will try and see if there is a clear point where overfitting happens and where the balance between training time and increase in testing score lies.

Learning rate

The learning rate determine how big of a change will be made in each training step. If the learning rate is too low the model might never have time to reach a point where the scores will improve. If the rate is too high it might overshoot it and never converge. In this experiment I will try and find the most ideal learning rate.

4.4.3 Best combination of parameters

In the last experiment I will use the information I gained from the previous experiments to select the best parameter values to try and get the best results out of the CNN model.
5 Results and discussion

5.1 Results for different parameter settings

5.1.1 Data content

The highest scores across the board was for the combination Summary and Profession. Profession by itself actually scores almost as high as when combined with Summary. Adding Answer seems to lower the score, which might mean that the query used at the event does not relate to the likelihood of a match.

<table>
<thead>
<tr>
<th>Part of data</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>0.67 (0.02)</td>
<td>0.71 (0.02)</td>
<td>0.63 (0.03)</td>
<td>0.66 (0.02)</td>
</tr>
<tr>
<td>Answer</td>
<td>0.67 (0.01)</td>
<td>0.66 (0.03)</td>
<td>0.67 (0.02)</td>
<td>0.67 (0.02)</td>
</tr>
<tr>
<td>Profession</td>
<td>0.76 (0.02)</td>
<td>0.80 (0.02)</td>
<td>0.75 (0.02)</td>
<td>0.77 (0.02)</td>
</tr>
<tr>
<td>Summary, Answer</td>
<td>0.75 (0.02)</td>
<td>0.75 (0.02)</td>
<td>0.71 (0.04)</td>
<td>0.73 (0.03)</td>
</tr>
<tr>
<td>Summary, Profession</td>
<td>0.78 (0.01)</td>
<td>0.81 (0.02)</td>
<td>0.76 (0.02)</td>
<td>0.78 (0.01)</td>
</tr>
<tr>
<td>Answer, Profession</td>
<td>0.74 (0.02)</td>
<td>0.75 (0.02)</td>
<td>0.71 (0.03)</td>
<td>0.73 (0.02)</td>
</tr>
<tr>
<td>Summary, Answer, Profession</td>
<td>0.73 (0.02)</td>
<td>0.70 (0.01)</td>
<td>0.74 (0.03)</td>
<td>0.72 (0.02)</td>
</tr>
</tbody>
</table>

Table 5.1: Mean scores with standard error for 5 independent measurements at each setting, which were the default settings but different parts of the data were used.
5.1.2 Positive and negative pair ratios

The ratio between positive and negative pairs in the training data seems to have the biggest effect on precision and recall. With more negative pairs the model is less likely to mark a pair as a match, however it seems that the pairs it does mark as a match has a higher chance of being right. The opposite happens when there are more positive pairs. This time the gain in recall is greater than the loss in precision meaning the F1-score is increases compared to the other two ratios. Which setting you chose depends on if finding some good matches and few to none bad matches is more important than finding many good matches at the cost of many additional bad matches is more import.

<table>
<thead>
<tr>
<th>Pos/neg ratios</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train - Test</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:1 - 1:1</td>
<td>0.73 (0.02)</td>
<td>0.7 (0.01)</td>
<td>0.74 (0.03)</td>
<td>0.72 (0.02)</td>
</tr>
<tr>
<td>1:2 - 1:1</td>
<td>0.68 (0.03)</td>
<td>0.79 (0.04)</td>
<td>0.5 (0.05)</td>
<td>0.61 (0.04)</td>
</tr>
<tr>
<td>1:1/2 - 1:1</td>
<td>0.69 (0.01)</td>
<td>0.63 (0.01)</td>
<td>0.91 (0.02)</td>
<td>0.75 (0.01)</td>
</tr>
</tbody>
</table>

Table 5.2: Mean scores with standard error for 5 independent measurements at each setting, which were the default settings but different ratios of positive and negative pairs were used for the training data.
When the experiment was repeated but all remaining negative pairs were put into the testing data the precision dropped radically. Recall behaved pretty much the same as for when the ratio was 1:1 for the testing data. When all the negative pairs were divided evenly between the training and testing data the model stopped tagging any pairs as a match. This makes accuracy very high, but only because True Negative drastically outnumber False Negative.

The experiments on large number of negative pairs most closely correspond to the real application of the model. This shows that gaining high level of precision on a 1:1 positive-negative ratio testing data is not enough for the model to be useful. It must be able to correctly pick out positive matches even when they are extremely spread out among a large number of negative pairs, and to do this without getting too many False Positives.

<table>
<thead>
<tr>
<th>Pos/neg ratios</th>
<th>Train - Test</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:1:1 Remaining</td>
<td></td>
<td>0.67 (0.02)</td>
<td>0.02 (0.00)</td>
<td>0.79 (0.01)</td>
<td>0.04 (0.00)</td>
</tr>
<tr>
<td>1:2:1 Remaining</td>
<td></td>
<td>0.9 (0.01)</td>
<td>0.05 (0.00)</td>
<td>0.55 (0.04)</td>
<td>0.09 (0.01)</td>
</tr>
<tr>
<td>1:1/2:1 Remaining</td>
<td></td>
<td>0.38 (0.01)</td>
<td>0.01 (0.00)</td>
<td>0.9 (0.03)</td>
<td>0.02 (0.00)</td>
</tr>
<tr>
<td>1:1:Half Remaining</td>
<td></td>
<td>0.98 (0.00)</td>
<td>Undefined</td>
<td>0 (0.00)</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

Table 5.3: Same as 5.2 but all remaining negative pairs were put into the testing data. One additional test was done were all negative pairs were split equally between the training and testing data.
5.1.3 Training and testing data ratios

When the amount of available data is this small you want to use as much as possible of it for training. The trade off is that the testing becomes unreliable when the testing data is too small. This trend can be seen in the results since the standard error decreases when the testing data increases. This effect is not very significant however and it seems like you can get away with a 90%/10% split as I used as default.

<table>
<thead>
<tr>
<th>% withheld for testing</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.73 (0.02)</td>
<td>0.7 (0.01)</td>
<td>0.74 (0.03)</td>
<td>0.72 (0.02)</td>
</tr>
<tr>
<td>30</td>
<td>0.72 (0.01)</td>
<td>0.71 (0.02)</td>
<td>0.73 (0.01)</td>
<td>0.72 (0.01)</td>
</tr>
<tr>
<td>50</td>
<td>0.71 (0.00)</td>
<td>0.73 (0.02)</td>
<td>0.68 (0.01)</td>
<td>0.7 (0.00)</td>
</tr>
</tbody>
</table>

Table 5.4: Mean scores with standard error for 5 independent measurements at each setting, which were the default settings but different percentages of the data was withheld for testing.
5.1.4 Filter size

The results are not very clear. It seems that 1-gram and 2-gram features have a negative impact on precision but improves recall, however the standard errors overlap too much between the different filter sizes to draw any definite conclusions. There does not seem to be a big benefit to increasing the filter sizes to capture relations that are further apart in the sentences either, instead most relevant features seems to be around the 3-gram range.

<table>
<thead>
<tr>
<th>Filter sizes</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3</td>
<td>0.73 (0.01)</td>
<td>0.68 (0.02)</td>
<td>0.77 (0.02)</td>
<td>0.72 (0.02)</td>
</tr>
<tr>
<td>3, 5, 7</td>
<td>0.73 (0.02)</td>
<td>0.7 (0.01)</td>
<td>0.74 (0.03)</td>
<td>0.72 (0.02)</td>
</tr>
<tr>
<td>3, 6, 9</td>
<td>0.72 (0.03)</td>
<td>0.7 (0.03)</td>
<td>0.74 (0.03)</td>
<td>0.71 (0.03)</td>
</tr>
</tbody>
</table>

Table 5.5: Mean scores with standard error for 5 independent measurements at each setting, which were the default settings but with different filter sizes.
5.1.5 Number of filters

The general trend for the number of filters seems to suggest a logarithmic increase in all scores for increasing number, except for the one off data point at 1 filter for precision. This spike might have something to do with a very low amount of pairs tagged as positive matches, a similar trend as we saw for when training on a large ratio of negative pairs, where the tipping point lies somewhere between 8 and 1 filters. The logarithmic trend line might be an over simplification however as it seems all scores level out at 32 filters. There does not seem to be a benefit for going over this number.

<table>
<thead>
<tr>
<th>No. of filters</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.69 (0.01)</td>
<td>0.73 (0.03)</td>
<td>0.64 (0.04)</td>
<td>0.58 (0.03)</td>
</tr>
<tr>
<td>8</td>
<td>0.71 (0.02)</td>
<td>0.68 (0.02)</td>
<td>0.73 (0.03)</td>
<td>0.71 (0.02)</td>
</tr>
<tr>
<td>16</td>
<td>0.69 (0.02)</td>
<td>0.71 (0.03)</td>
<td>0.7 (0.03)</td>
<td>0.7 (0.03)</td>
</tr>
<tr>
<td>32</td>
<td>0.73 (0.02)</td>
<td>0.7 (0.01)</td>
<td>0.74 (0.03)</td>
<td>0.72 (0.02)</td>
</tr>
<tr>
<td>64</td>
<td>0.73 (0.01)</td>
<td>0.71 (0.02)</td>
<td>0.72 (0.01)</td>
<td>0.72 (0.01)</td>
</tr>
</tbody>
</table>

Table 5.6: Mean scores with standard error for 5 independent measurements at each setting, which were the default settings but with different number of filters.
Figure 5.6: The x-axis shows number of filters on a logarithmic scale. Mean scores with standard error as error bars for 5 independent measurements at each setting, and with a logarithmic trend line fitted to the data points.

5.1.6 Number of neurons

It seems pretty clear from the results that somewhere around 256 neurons are optimal for this model and data. Perhaps less neurons cannot capture all necessary features and more neurons will dilute the features too much. Maybe if we were to use a more feature rich data set the model could benefit from more neurons.

<table>
<thead>
<tr>
<th>No. of neurons</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>0.74 (0.02)</td>
<td>0.73 (0.02)</td>
<td>0.74 (0.02)</td>
<td>0.73 (0.01)</td>
</tr>
<tr>
<td>256</td>
<td>0.76 (0.02)</td>
<td>0.74 (0.02)</td>
<td>0.77 (0.03)</td>
<td>0.75 (0.02)</td>
</tr>
<tr>
<td>512</td>
<td>0.75 (0.02)</td>
<td>0.74 (0.03)</td>
<td>0.77 (0.02)</td>
<td>0.75 (0.02)</td>
</tr>
<tr>
<td>1024</td>
<td>0.73 (0.02)</td>
<td>0.7 (0.01)</td>
<td>0.74 (0.03)</td>
<td>0.72 (0.02)</td>
</tr>
<tr>
<td>2048</td>
<td>0.72 (0.02)</td>
<td>0.69 (0.02)</td>
<td>0.75 (0.02)</td>
<td>0.72 (0.02)</td>
</tr>
</tbody>
</table>

Table 5.7: Mean scores with standard error for 5 independent measurements at each setting, which were the default settings but with different number of neurons.
5.1.7 Dropout

There seems to be little to no benefit in using dropout for this model and data, although the overlapping standard errors make it hard to say for sure. The results might be different when using many more steps while training, but as seen in 5.1.9 there does not seem to be any signs of overfitting even at 10,000 steps.

Table 5.8: Mean scores with standard error for 5 independent measurements at each setting, which were the default settings but with different rates of dropout.
5.1.8 Batch size

It is hard to draw any conclusions from the results due to the overlapping standard errors for all data points. It does not seem to be a great benefit to increasing the batch size, so if training time is important you do not seem to sacrifice much by lowering the batch size.

<table>
<thead>
<tr>
<th>Batch size</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.74 (0.02)</td>
<td>0.75 (0.03)</td>
<td>0.73 (0.03)</td>
<td>0.74 (0.02)</td>
</tr>
<tr>
<td>50</td>
<td>0.73 (0.02)</td>
<td>0.7 (0.01)</td>
<td>0.74 (0.03)</td>
<td>0.72 (0.02)</td>
</tr>
<tr>
<td>100</td>
<td>0.74 (0.03)</td>
<td>0.76 (0.05)</td>
<td>0.75 (0.03)</td>
<td>0.75 (0.03)</td>
</tr>
</tbody>
</table>

Table 5.9: Mean scores with standard error for 5 independent measurements at each setting, which were the default settings but with different batch sizes.
Figure 5.9: The x-axis shows the batch size. Mean scores with standard error as error bars for 5 independent measurements at each setting, and with a linear trend line fitted to the data points.

5.1.9 Number of steps

The results show an increase to all scores when increasing the number of steps. Even at 10 000 steps the trend does not seem to shift due to overfitting, showing that there might be a benefit to going even higher. This however must be weighed against the increased training time since a doubling in steps also doubles the training time.

<table>
<thead>
<tr>
<th>Steps</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.66 (0.02)</td>
<td>0.66 (0.03)</td>
<td>0.71 (0.05)</td>
<td>0.67 (0.01)</td>
</tr>
<tr>
<td>1000</td>
<td>0.73 (0.02)</td>
<td>0.7 (0.01)</td>
<td>0.74 (0.03)</td>
<td>0.72 (0.02)</td>
</tr>
<tr>
<td>10000</td>
<td>0.77 (0.01)</td>
<td>0.77 (0.01)</td>
<td>0.77 (0.01)</td>
<td>0.77 (0.01)</td>
</tr>
</tbody>
</table>

Table 5.10: Mean scores with standard error for 5 independent measurements at each setting, which were the default settings but with different number of steps.
Figure 5.10: The x-axis shows the number of steps. Mean scores with standard error as error bars for 5 independent measurements at each setting, and with a linear trend line fitted to the data points.

5.1.10 Learning rate

The effect of the learning rate is not very straightforward and it seems that the default learning rate of 0.01 is right around the optimal number.

<table>
<thead>
<tr>
<th>Learning rate</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.73 (0.03)</td>
<td>0.70 (0.04)</td>
<td>0.77 (0.05)</td>
<td>0.73 (0.03)</td>
</tr>
<tr>
<td>0.01</td>
<td>0.73 (0.02)</td>
<td>0.70 (0.01)</td>
<td>0.74 (0.03)</td>
<td>0.72 (0.02)</td>
</tr>
<tr>
<td>0.001</td>
<td>0.64 (0.03)</td>
<td>0.78 (0.05)</td>
<td>0.52 (0.07)</td>
<td>0.59 (0.05)</td>
</tr>
</tbody>
</table>

Table 5.11: The x-axis shows the number of steps. Mean scores with standard error as error bars for 5 independent measurements at each setting, and with a linear trend line fitted to the data points.
5.2 Final results

After investigating the effects of tuning different parameters I have determined the settings most likely to give the highest scores. For part of data I use Profession and Summary; the positive:negative ratio of pairs in the training data is 1:1; 90% of the data is used for training and 10% for testing; the filter sizes are 3, 5 and 7; the number of filters is 32; the dense layer has 256 neurons; there is no dropout; the batch size is 10; the number of steps is 10 000; the learning rate is 0.01. The results for these settings compared to the default settings can be seen in Table 12.

<table>
<thead>
<tr>
<th>Settings</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>0.73 (0.02)</td>
<td>0.70 (0.01)</td>
<td>0.74 (0.03)</td>
<td>0.72 (0.02)</td>
</tr>
<tr>
<td>Best settings</td>
<td>0.83 (0.02)</td>
<td>0.83 (0.02)</td>
<td>0.82 (0.03)</td>
<td>0.82 (0.02)</td>
</tr>
</tbody>
</table>

Table 5.12: Mean scores with standard error for 5 independent measurements. Parameters were set to optimize the scores.
Figure 5.12: TF-IDF scores for the three events: top left is event 1, top right is event 2 and bottom left is event 3.

5.12 shows the original TF-IDF results created by Lunchback for the three events. The x-axis indicates the similarity between the two users in a pair according to the TF-IDF algorithm, y-axis indicates the number of pairs. The matched and non-matched pairs are represented respectively by blue and red. From the figures above, we can easily see that most matched and non-matched pairs are overlapping between roughly scores 0.3-0.7, it basically states that the TF-IDF model cannot distinguish between matched and non-matched pairs and would therefore perform with an accuracy of about 0.5, same as chance. An ideal version for the TF-IDF model should show that two groups are separated from each other and stay on the two opposite sides of the x-axis.

Compared to the TF-IDF model, the CNN model can identify if a pair is matched with over 73% accuracy with the default settings, and the one with best settings even reached 83% accuracy. This declares that the CNN model outperformed the TF-IDF model. Additionally, the final results include measurements for precision, recall and F1-score which can be used for future improvements of the model.
6 Conclusion and future work

Even with the limited data sets used in the experiments, the three research questions can still be answered. First, evidently CNN model can be used for semantic text matching. Second, CNN model can be greatly improved by modifying input data sets and tuning multiple parameters in the CNN structure. Finally, the CNN model in this thesis significantly outperformed the old TF-IDF model from Lunchback.

For future works there is a couple of areas which could be explored further. First is to update the word vectors during training to better fit them to the training data. Second is better preprocessing of the data to fit the structure of the word vectors since currently words which are not represented in the word vectors are ignored. Third is to gather a larger and higher quality data set. Lunchback is currently developing a web game in which people can help to manually match users. The users will also answer a more focused set of queries which will hopefully generate more future rich data. Finally the full impact of all parameters were not explored in this thesis and further improvements can surely be made.
Bibliography


