Friedrich-Alexander-Universität Erlangen

Bachelor thesis

Professur für Wissensrepresentation und -verarbeitung

Using machine learning to support annotating of keywords in mathematical texts

Author:
Marian Plivelic

Supervisor:
Prof. Dr. Michael Kohlhase

Erlangen, 1. Februar 2019

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 1. Februar 2019

Contents

1 Introduction 2
1.1 Motivation & Overview 2
1.2 Related Work 2

2 Preliminaries 3
2.1 Python & Keras/TensorFlow 3
2.2 SMGloM 3

3 Feature Selection 4
3.1 Tokens & Labels 4
3.2 Embedding 4
3.2.1 Training on SMGloM 5
3.2.2 FastText 5
3.2.3 GLoVe 5
3.3 Wikify! 5
3.3.1 tf-idf 6
3.3.2 χ^2-test 6
3.3.3 Keyphraseness 6
3.3.4 Calculating the features for training 7
3.4 Part-of-speech tags 8

4 Models 8
4.1 Shallow Models 8
4.2 Hidden Markov Model 9
4.3 Deep Model 10
4.4 Evaluation & Comparison 11
4.4.1 Binary VS. Ternary predictions 11
4.4.2 Shallow Models 13
4.4.3 Hidden Markov Model 13
4.4.4 Deep Model & Feature importance 13
4.4.5 Expected workload reduction 15

5 Integration 16
5.1 Tagging 16
5.2 Auto-Completion 16
5.3 Linting 17
5.4 Goto operation & Import graph 18

6 Conclusion 19

7 Images 20

1 Introduction

1.1 Motivation & Overview

With the mountain of papers being released each day, understanding scientific texts is becoming more and more difficult and frankly impossible for humans. New ways of analyzing all the data are slowly becoming required for finding relevant data. In order to aid the human researcher with extracting relevant information, I propose a program that extracts phrases in scientific text, which might either be something used to define a concept, called *trefi*, or something that is defined using these *trefis*, called a *defi*. A user can use these automatically generated relevant phrases and create new annotations faster.

```
A prime number is a natural number greater than 1
that has no positive divisor other than 1 and itself.
```

In this example the phrase *prime number* is a *defi* using the *trefis natural numbers* and *divisor*.

Important for the extracted phrases is that the person, who creates annotations, is able to trust the model to catch **all** relevant phrases, while keeping it clean from irrelevant phrases as much as possible. This allows the user to be sure, that all automatically highlighted portions of the text contain the relevant information needed to create annotations, increasing productivity in the process. If not all relevant phrases are caught, then the whole program would be useless, because the human still would have to read everything.

My target scores for this reason are a recall as close to 100% as possible, while also keeping precision as high as possible, but it doesn't have to be at a 100%. I want to essentially keep false negatives at 0% while trying to minimize false positives as much as possible.

To achieve these scores I worked out some features, that help the annotation tool to analyze the dataset (section 3). With these features then, I explain how the generation model works (section 4.3), and how it compares (section 4.4) to simpler models (section 4.1).

Besides automated annotation generation, which I will discuss in section 4, there needs to be an intuitive way to use the tagger. I integrate the model into an editor extension that displays the generated annotations in an easy to understand manner (section 5). And if the program is implemented in an editor, we might as well implement some staples of modern editors to increase productivity even more, like autocompletion (section 5.2) and linting (section 5.3).

1.2 Related Work

This sounds simple enough, but extracting meaning from scientific texts is hard. Decades of research have already been done using grammars[Zin04], rule-based spotters[Rab17; Sch16] and other classifiers utilizing features extracted from a lot of data[MC07] instead of tokens. These approaches do work rather well, but developing such algorithms is complicated and requires a lot of feature engineering in order to make proper predictions. My approach will

be circumventing the difficult feature engineering step by using deep learning methods, that are able to learn features from raw text.

2 Preliminaries

2.1 Python & Keras/TensorFlow

With the recent popularity of deep learning, a lot of libraries have been developed in the past years. Especially python with it's dynamic typing has been a popular programming language for rapid development and with the open source deep-learning library TensorFlow[Aba+15] I am able to use the power of my GPU from inside python.

On top of that I use Keras[Cho+15] which provides a less verbose interface for common deep-learning layers than TensorFlow. Other utilities like metrics and methods for constructing, testing and debugging models are also available.

2.2 SMGloM

I use the Semantic Multilingual Glossary for Mathematics (SMGloM[Gin+16]) as the dataset used for training the machine learning algorithms. SMGloM is, as suggested by the name, a Glossary of Mathematical definitions, with hand-made annotations indicating what is being defined (called a *definiendum* or *defi*) and annotations indicating what is used for that definition (called a *trefi*). It also allows to define concepts in several languages, and it has an online interface.

A sample from the dataset looks like this:

```
\begin{definition}
    The \defiii{category}{of}{small categories}
    (denoted as $\CatCategory$) has all
    \trefi{small} \mtrefi{categories} as
    \trefis{object} and \trefis{functor}
    as \trefis{arrow}.
\end{definition}
```

In this example you can see, that the definiendum **category of small categories** is tagged with \defi. **small**, **categories**, **object**, **functor** and **arrow** are tagged as \trefi.

After parsing the files, I can use the text and labels to train machine learning algorithms to predict new annotations.

3 Feature Selection

Even though deep-learning should be able to handle the tokens directly, adding some simple features has good chances of increasing the overall performance. In this section I will discuss all features and feature transforms used for training and evaluate their performance in section 4.4.

3.1 Tokens & Labels

The most important feature are the tokens themselves.

From the example shown in the preliminaries section (2.2), using a custom version of an existing python LaTeX parser[Wan16], I can extract these tokens with their labels:

The	text
category	defi
of	defi
small	defi
categories	defi
(	text
denoted	text
...	...
functor	trefi
as	text
arrow	trefi
.	text

Doing this for the whole dataset I get about 36167 (non-unique) tokens in 852 files, with each file being one sample as most only contain one definition. 11% of these tokens are labeled as *trefi*, further 11% as *defi*. 88% of tokens have no label, which makes the classes unbalanced. This is an problem, but applying class weights using log-probabilities this isssue is mostly solved.

The log-probability weight at occurence rates of 88%, 11% and 11% are 0.19, 2.45 and 2.45 respectively for *text*, *trefi*, *defi*. If we unite *trefis* and *defis* for a total of 22% of *keywords* (see 4.4.1), the log-probability weighting becomes 0.3 for *text* and 1.76 for the *keyword* label.

The log-probability is given by $\ell_{K,D}(k) = -log(\frac{count_D(k)}{\sum_{i=0}^{K} count_D(i)})$, where K are the classes and $count_D$ returns the number of phrases of class k, i in the data set D.

3.2 Embedding

Because machine learning algorithms can't work with strings directly, some mapping from tokens to vectors of numbers is required for most machine learning applications. This transformation of the original tokens into a number vector is called an embedding.

There are many ways to do this, but the base idea is, that ”a word is characterized by the company it keeps” (John R. Firth). That means, that an embedding is a representation for a word, that let’s you estimate the words it co-occurs with (or the inverse). In this section I explain, what embedding methods I considered and what I ended up using.

3.2.1 Training on SMGloM

Creating a new embedding is a possibility, but because SMGloM only has about 1000 unique tokens, training new embeddings on the training set does not perform well on unseen data, as my dataset only has around 1000 unique tokens.

3.2.2 FastText

One popular embedding is FastText[Jou+16]. It has the advantage, that it can estimate embeddings for unseen words using n-grams. But I could not use it in my research as dimensionality reduction on the n-grams doesn’t seem possible and I don’t have enough computing power to create a new low-dimensionality model from the 60GB Wikipedia dump. But without dimensionality reduction using a 300 dimensional embedding for a 1000 word training set doesn’t work well.

3.2.3 GLoVe

Instead I use pre-computed GLoVe[PSM14] embeddings. I create the embedding layer by loading the 50D glove embedding[JP14] trained on 400.000 tokens, then applying principal component analysis (PCA) to lower the dimensionality to 10D in order to reduce the number of weights the network has to learn. This also introduces aliasing between learned and unseen words, increasing generalizability. There exist also pre-computed FastText embeddings[Mik+18], just like the GLoVe embeddings that should perform equally well.

3.3 Wikify!

Besides the embeddings, I use the three features showcased in the Wikify![MC07] paper for automated wikification of wikipedia articles, which are a χ^2-test, *tf-idf* and *keyphraseness*. The paper selects these features in order to ”allow readers to easily and conveniently follow their curiosity or research to other articles.” They also selected these features, because they find phrases to annotate that do not conflict with the guidelines wikipedia has for the amount of links and what to link. I do not have any discernible restrictions for my annotations, except that they should mirror the annotations of the probability-density-function (PDf) of the labels provided by the data set, but that should not be of concern right now.

3.3.1 tf-idf

The *term-frequency-inverse-document-frequency*[Ram03] measure, or *tf-idf* for short, is a measure for importance of words in NLP.

Given a word w, document d and all documents in the corpus D, the tf-idf function returns higher scores as the term frequency $f_{w,d}$ increases and lower scores as the corpus frequency $f_{w,D}$ increases. The term frequency is the count of how often the term w appears in the document. The document frequency is the count of in how many documents of the corpus the term appears in.

$$tfidf : f_{w,d} \times log_e(\frac{|D|}{f_{w,D}})$$

This results in common words like 'The', and 'I' to have low scores, while uncommon words like 'vector' or 'logarithm' will receive a high score.

Note, that for my implementation I normalized $f_{w,d}$ by the number of words in the document d, and used log_2 instead of log_e in the *idf*-term, because it appeared to work better. A reason normalization works better, may be because the relative relation is more important, than the absolute relation.

3.3.2 χ^2-test

The χ^2-test is described with this contingency table:

count (phrase in document)	count (all other phrases in document)
count (phrase in other documents)	count (all other phrases in all other documents)

The idea behind this is to calculate the independence between phrase count in the current document and the rest of the corpus. This means, that a high degree of independence indicates a phrase, that was probably placed there on purpose and is more likely to be a relevant phrase.

3.3.3 Keyphraseness

At last, the paper introduces the *keyphraseness* measure.

$$P(keyword|w) \approx \frac{count(D_{key})}{count(D_w)}$$

This score was in the paper as well as in my case the best performing measure. The *keyphraseness* of a word w is the probability that the word is a keyword. This probability can be approximated by dividing the number of times the word w was used as a keyword D_{key} in all

documents of the corpus D by the count it was not used as a keyword D_w. In the context of SMGloM, I define keywords (see 4.4.1) to be all *trefis* and *defis*. I do not make a distinction between the tags.

This is a good measure for my problem, because it incorporates our knowledge base as a single feature in our model, letting the models know, if a phrase was already defined in the corpus, making it possible to accurately tag it as an *trefi*.

3.3.4 Calculating the features for training

The formulas for the features have been established, but there are still problems when it comes to using them for training.

Problem 1: These features make heavy use of the corpus as a whole. A problem occurs, when a phrase or word of a new sample is not part of the corpus. All three features break because of division by 0, or in the case of χ^2, a too low sample count. I therefore just set the value for that word to 0. This is again a noise factor for that I don't know any good solutions. Maybe providing a "word missing" feature might help, but I did not test that here, and just hope, that the model deals with this by learning that a flat 0 means "missing or unimportant."

Problem 2: The second problem is, that each sample must not know about the labels and words in itself in order to not skew the values for a label or word that would be newly introduced with the new sample.

The wikify features for a sample $x \in D$ from the corpus D are calculated by creating a temporary corpus $D' = \{y \in D | x \neq y\}$ for each sample and applying the three functions using that new corpus instead.

3.4 Part-of-speech tags

I also make use of part-of-speech tags, or POS tags for short.

POS tags essentially indicate what class of word each word is. I use the NLP library *nltk*[LB02] to generate the tags.

A tagged sentence looks like something like this:

Word	And	now	for	something	completely	different	.
POS tag	CC	RB	IN	NN	RB	JJ	.

With this information, the network should be able to learn, that for example, the tags "CC", "RB" and "IN" are usually not words relevant for our goal. A pro for POS tags is, that even if the word is unknown, the POS tag is not, yielding better results.

A problem is, that latex source code is not the best target for POS tag generation as latex contains a lot of markup tokens. For example the "&" in tabular environments and all math environments in general. There isn't too much that can be done without complex replacement rules and even after replacing math environments with a special "MathFormula" token or similar, will nltk sometimes not be able to create a fitting list of POS tags for a sentence resulting in some noise.

For training purposes I represent the POS tags as a 44 dimensional one-hot encoded bag-of-words like vector per word in the input sequence, where each dimension represents the presence of a part-of-speech tag.

4 Models

Now that dataset and features have been discussed, I want to first justify the use of deep-learning for this task by showing the performance of shallow learning algorithms, as well as Hidden Markov Models. After introducing the the deep model and explaining how all the models were trained, I evaluate them in section 4.4.

The settings all models share are, that the train/validation split was 80%/20% and the mean/deviation was calculated by running the models 5 times.

4.1 Shallow Models

I trained various shallow models using the implementations from the python library sklearn [Ped+11].

Shallow models are of course not a solution to this NLP problem, but I wanted to have a reference for how they perform on this task in order to justify computation heavy deep-learning methods.

The models were k-nearest neighbors (kNN), decision trees, random forests using decision trees, AdaBoost with decision trees, gaussian naive bayes (GaussianNB) and quadratic discriminant analysis (QDA).

Because none of these algorithms are suitable for time series, I trained them on a randomly selected fixed time window size of 4 time steps. Every model was trained using the token, tf-idf and keyphraseness features.

4.2 Hidden Markov Model

Hidden Markov Models[RJ86] are a popular algorithm for classifying time series. The original algorithm only allows for unsupervised training, but when classifying something like POS tags or keywords, you can and need to train the markov model in a supervised fashion.

The supervised algorithm[Neu] works by assuming, that the labels Y of the dataset are the hidden state y and the samples are the observations X.

Then define the initial state probabilities to be the number of times the state y is at the beginning of an observation sequence:

$$P(y) = \frac{|\{seq \in Y | seq[0] = y\}|}{count(y)}$$

And transition probabilities from the last state y_{i-1} to the new state y_i to be the number of times the state y_{i-1} is the predecessor to the state y_i, divided by the number of times y_{i-1} occurs in the corpus.

$$P_T(y_i|y_{i-1}) = \frac{count(y_i|y_{i-1})}{count(y_{i-1})}$$

The emission probabilities for the word x_i given current state y_i as:

$$P_E(x_i|y_i) = \frac{count(x_i|y_i)}{count(y_i)}$$

Then for a given sequence probability for a state sequence Y is:

$$P(Y) = P(y_0) \prod_{i=1}^{|Y|} P_T(y_i|y_{i-1})$$

here y_0 is the initial state.

The associated probability of a sequence of words X for the given states Y is:

$$P(X|Y) = \prod_{i=0}^{|Y|} P_E(x_i|y_i)$$

Using the viterbi algorithm[For73], you can estimate the most likely state sequence Y for the observations X.

4.3 Deep Model

In order to keep it simple, I used the most basic deep model possible (figure 1). It consists of the input tokens being embedded, then concatenated with the other features. The sample is then simply fed through two layers of 32 unit bidirectional GRUs, followed by two layers of 32 unit dense layers, that feed everything into the final dense layer for classification with the softmax loss. The number of units in the last layer changes depending on the number of different annotations being predicted. So it is either 2 units for predicting *text* and *keyword*, or 3 units for further discriminating *keyword* into *trefi* and *defi*.

The embedding layer maps the tokens to 10 dimensional vectors of floats, as described in section for embeddings (3.2).

The GRU units are bidirectional and use tanh as activation, sigmoid as recurrent activation. They also have a recurrent dropout rate of 0.1 during training.

The dense layers have sigmoid as activation and a dropout of 0.5.

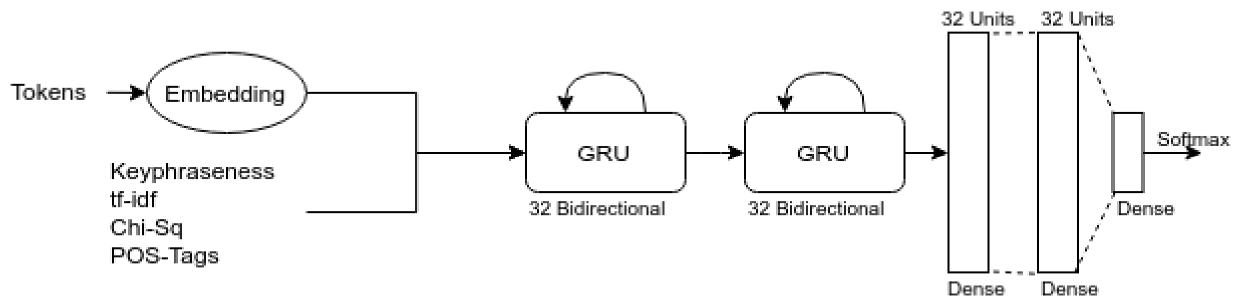

Figure 1: The basic deep model architecture used. Inputs get concatenated, fed into a few GRUs and are then classified by three dense layers.

I trained using the ADAM optimizer[KB14] with default keras settings and early stopping after the training loss appears to have converged for 5 epochs. The average epoch count was at around 60, which takes about 10 minutes to train on my NVIDIA GeForce GTX 1060 6GB. An example training history is shown in figure 2.

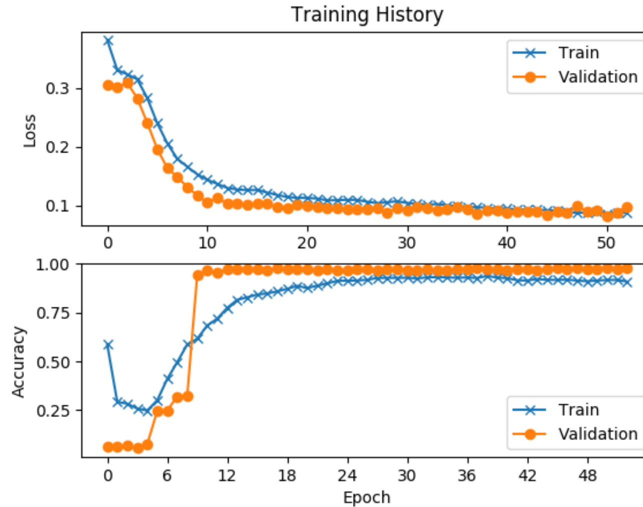

Figure 2: Example of the training history of one model. On top is the training loss vs. validation loss and on the bottom is the training accuracy vs validation accuracy. The x-axis shows the number of epochs trained. Both x-axes have the same number of epochs even though the displayed count is different. The validation loss is lower than the training loss, because of regularization during training.

4.4 Evaluation & Comparison

In this section I evaluate the binary and ternary types of predictions (4.4.1), show the performance of all models (4.4.2, 4.4.3) as well as the importance of the features (4.4.4) in order to find the best deep model for the task. After that, I will analyze and evaluate how good the model is in terms of reducing the actual amount of text required to work through (4.4.5).

I evaluate the models using the measures accuracy, F1, recall and precision. Accuracy is the percentage of true-positives vs. false-negatives. Recall is the percentage of how many relevant data points have been selected. Precision is the percentage of relevant vs. not relevant data points selected. F1 is the weighted average $2 \times \frac{recall \times precision}{recall + precision}$ of those two. All scores are better the higher they are.

4.4.1 Binary VS. Ternary predictions

I have already mentioned that maybe using the full dataset with *trefi* and *defi* labels might not be a good idea. Instead uniting them and calling it a *keyword* label might be better approach for the learning algorithms.

Computing the confusion matrices for two identical models (figures 3, 4), with the only difference being that the left one (figure 3) predicts *text* and *keyword*, and the right one (figure 4) predicts *trefi* and *defi* instead of *keyword*.

On the y axis are the true labels from the dataset and on the x axis are the labels the models

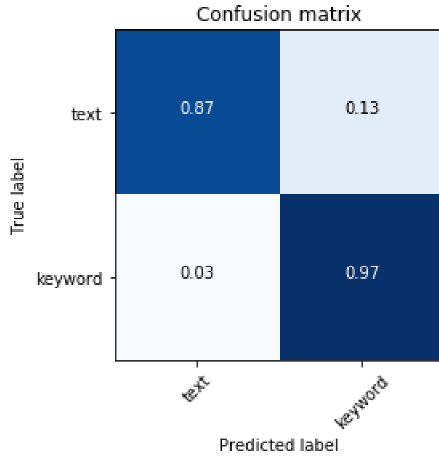

Figure 3: Binary confusion matrix

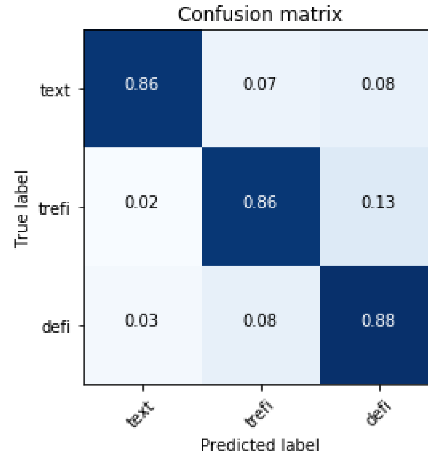

Figure 4: Ternary confusion matrix

predicted. On the diagonals are the accuracies of for each label. The cells below the diagonal contain false negatives, while the cells above the diagonal contain false positives.

You can see that the binary model has 3% false-negatives and 13% false-positives. The ternary model has $2\% + 3\% = 5\%$ false-negatives and $7\% + 8\% = 13\%$ false-positives.

This shows that both types of models are more or less equal in their ability to separate irrelevant text from relevant keywords. But the ternary model has some additional internal confusion between *trefis* and *defis*. It fails to predict 8% of *defis* and 13% of *trefis* causing it to have a lower mean accuracy.

In practice, this error is visible as swapped predictions. For example figure 5 shows that the model swapped the *trefi* (blue) and *defi* (red) label.

A magic square is an arrangement of numbers

Figure 5: Model misclassifies and swaps *defi* (red) with *trefi* (blue)

And the uncertain predictions, that are close to 50% between *trefi* and *defi* might cause the model to switch the label mid-phrase. Figure 6 shows that the phrase *balanced prime* and *arithmetic mean* do not have uniform labels attached.

A balanced prime is a prime number
that is equal to the arithmetic mean

Figure 6: Model predicts inconsistent labels

This is a problem for the person writing annotations, because it requires further reading of the context. The tagger, that wants to aid the user to easily identify phrases that might need annotating, has to be able to deal with that as well.

But there is an easy solution. By simply merging the labels to a single *keyword* label. This allows the tagger to display all keywords as *trefi* if there is a *defi* found in the corpus, that

the newly labeled keyword might be referencing. All other identified keywords where no reference can be found, are then probably *defis* or false-positives.

For this reason, i will completely drop ternary prediction models from now on and all models are trained to predict *text* and *keyword* labels.

4.4.2 Shallow Models

While the accuracies of the models appear to be quite high (table 1), keep in mind that a classifier that only predicts *text* will have an accuracy of 88% as that portion of the training data set is labeled *text*. But such a classifier would also have a F1-Score of 0.

Algorithm	Accuracy	F1-Score	Recall	Precision
kNN	90±0%	0.73±0.01	73±1%	74±1%
Decision Tree	91±0%	0.76±0.00	75±0%	76±1%
Random Forest	**93±0%**	**0.79±0.00**	76±1%	**82±1%**
AdaBoost	91±0%	0.75±0.01	77±1%	73±1%
GaussianNB	89±1%	0.73±0.01	**79±0%**	68±2%
QDA	90±1%	0.74±0.01	78±0%	70±2%

Table 1: Evaluation of the various shallow algorithms.

As all models have a relatively high recall and precision of over 75%, which means that they did learn something and do not just predict *text*. Random forests using decision trees perform best on this dataset, but have a low recall. The following deep models will perform much better than this.

4.4.3 Hidden Markov Model

Markov Models have been proven to be quite useful for time series and many NLP tasks like POS-Tagging[Tou+03], so in theory they should perform relatively well on this task.

Model	Accuracy	F1-Score	Recall	Precision
HMM	90±0%	0.70±0.01	69±2%	70±2%

Table 2: Evaluation of the Hidden Markov Model

My implementation didn't perform well, but I only implemented bi-grams and did not include other important optimizations like smoothing etc. HMMs could, with proper optimizations, perform much better and could serve as a much more lightweight alternative to deep models, which are quite computationally expensive.

4.4.4 Deep Model & Feature importance

In order to find and evaluate the best deep model, we need to find it in all combinations of our five features.

Determining the importance of each feature is an essential step in optimizing the model with limited data and not enough time to test out all 120 combinations multiple times for mean and deviation.

Feature	Accuracy	F1-Score	Recall	Precision
Tokens	91±1%	0.52±0.02	**97±1%**	35±2%
tf-idf	25±10%	0.10±0.01	86±2%	5±1%
Keyphraseness	**95±1%**	**0.66±0.05**	90±2%	**53±6%**
χ^2	89±3%	0.42±0.04	88±1%	28±4%
POS tags	94±1%	0.64±0.02	95±0%	49±2%
Tok+tfidf+key+χ^2+POS	95±1%	0.65±0.07	95±1%	49±9%

Table 3: Scores for all single-feature deep models that predict the *keyword* label. And a model that uses all features at once.

Table 3 shows accuracy, F1-score, recall and precision of models, that were trained with only one feature from section 3. It is evident, that keyphraseness has the best scores overall, but tokens have the best recall.

The tf-idf feature stands out, as it does not seem to be able to predict anything on it's own.

Comparing these uni-feature models to a model that utilizes all the features (bottom of table 3), we observe that the model is not the best by any measure. The reason for this might be that there is not enough data to train the additional weights of the model. It might also be, that the capacity of the model was not high enough, but I tested it with increased capacity without being able to reach comparable results. I can conclude, that there is a model between one and all features used, that we have to find.

Feature	Accuracy	F1-Score	Recall	Precision
Tokens + tf-idf	92±1%	0.56±0.02	**96±1%**	39±2%
Tokens + Keyphraseness	**96±0%**	**0.73±0.02**	**96±1%**	**59±3%**
Tokens + χ^2	91±1%	0.53±0.01	**96±1%**	37±1%
Tokens + POS tags	95±1%	0.67±0.07	**96±1%**	52±8%
tf-idf + Keyphraseness	**96±1%**	0.68±0.02	90±1%	55±2%
tf-idf + χ^2	89±1%	0.43±0.03	87±1%	29±2%
tf-idf + POS tags	95±0%	0.63±0.02	**96±1%**	47±2%
Keyphraseness + χ^2	94±1%	0.63±0.01	93±2%	47±1%
Keyphraseness + POS tags	**96±0%**	0.07±0.01	**96±0%**	55±1%
χ^2 + POS tags	93±1%	0.55±0.03	95±2%	39±3%

Table 4: Scores of all deep-models that use two features and predict the *keyword* label.

In order to find it, I analyze the feature orthogonality by training models that use two features. Features that have equal information, will not perform better together while features that complement each other might make a good model.

After adding a feature, the models appear to have a reduced recall score, but they achieve a higher maximum f-score.

The tf-idf feature appeared to be useless if it is on it's own, but all combinations with tf-idf in it increase their F1-Score by at least 1%.

If we count, how often a feature performs best at a feature, we can get an idea of how good the feature truly is.

Feature	Times best
Tokens	8
Keyphraseness	7
POS tags	3
tf-idf	2
χ^2	1

Table 5: Counting how often a feature is in the best of a measure

After counting these scores in tables 3 and 4, we can create an order for the features (table 5). The most important feature are the tokens, then keyphraseness, POS tags, tf-idf and last place is the χ^2-test.

Features	Accuracy	F1-Score	Recall	Precision
Tokens + Keyphraseness + POS tags + tf-idf	**97±0%**	**0.75±0.03**	**97±0%**	60±3%
Tokens + Keyphraseness + POS tags + χ^2	94±1%	0.64±0.04	95±1%	48±5%
Tokens + Keyphraseness + POS tags	96±0%	**0.75±0.03**	96±1%	**61±3%**

Table 6: Evaluating the models that use the best features

Evaluating the models that keep tokens, keyphraseness and POS tags (table 6), I can conclude that using all features except χ^2 is probably the best model currently possible.

4.4.5 Expected workload reduction

In section 1.1 I explained why the recall is the most important feature, but of course, just searching for the model that reaches 100% is not optimal, as as it will get more and more false-positives until it only predicts keywords when it reaches 0% precision. By combining the number of keywords with the recall and precision scores, we are able to calculate how noisy the output of the model is. With the noise we can estimate how much the expected workload reduction is by using this machine learning approach.

As mentioned in section 2.2, there 22% of all tokens are labeled as keywords. That means, that roughly 22% of words in each file are keywords. Using the best model from section 4.4.4, we can calculate that if 97% (recall) of keywords are selected and 60% (precision) of the selected phrases are keywords, then there are 40% more keywords predicted than there actually are. Making for a total of about $22\% \times 140\% \approx 30\%$ automatically tagged tokens in each document. This translates to an reduced workload of about **70%**.

5 Integration

Now with the model, that is able to identify keywords, I can implement a server, that provides the functionality of the model. Because functionality servers are generally just command line applications and command line applications are not the best way to increase productivity I decided to integrate and implement it as an extension for the lightweight editor Visual Studio Code, because it has an easy to use interface, that allows me to visualize everything.

5.1 Tagging

The main feature of the extension is of course the annotation-hint generation functionality. I send the raw latex source currently open in the editor to the tagging-server, that is running in the background. The server than parses the file and sends back hints with location information, that then can be displayed (figure 7).

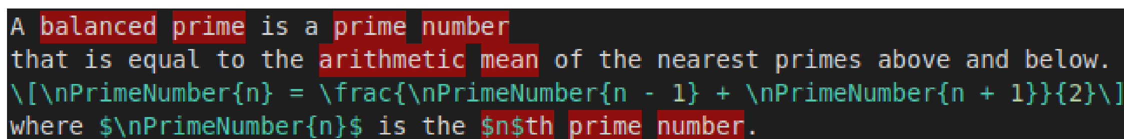

Figure 7: Generated annotation hints for the definition of balanced prime.

Then the hints can be used to show possible references. When viewing a definition (figure 7) and annotations have been generated, I can take adjacent ones, which are for example *balanced prime*, *prime number*, and search for definitions in the database. There is no definition for *balanced prime* so nothing needs to be done. A definition for *prime number* is available and the server can additionally suggest an annotation with the origin location of the definition.

5.2 Auto-Completion

Autocompletion is for many the most important feature of an editor. Not needing to remember exactly how to write something, or even finding mistakes by trying to type something only to see, that the desired symbol could not be found, because imports are missing probably, makes everything a lot faster.

I therefore added semantic autocompletion for all files and symbols required to create annotations for the SMGloM glossary.

- An import environment without an argument (figure 16) is provided with a list of all modules in the same directly as the current module.
- A list of all folders, which define any modules, is displayed if the user wants to import a module from another folder. (figure 17)

- The list of modules is filtered to only contain modules from the folder specified in the import argument. (figure 18)
- When creating a new language binding, all modules from the same folder are shown. (figure 19)
- The same is possible for *trefis*. When creating a new *trefi*, a list of modules is shown that can be referenced. (figure 20)

Each *defi* either has an explicitly defined *name=* argument or it is generated by concatenating all defined tokens by hyphens. I provide autocompletion for all instances where these names can be used.

- So of course, autocompletion for all names of imported *defis* is provided after a *name=* argument. (figure 21)
- For example, when directly referencing a specific *defi* inside the argument of a *trefi*. A list for all *defi* names defined in the module before the '?' is displayed. (figure 22)

And there is autocompletion for not only the symbols, but also the tokens, that define these symbols.

- Completes the text for the locally defined symbol. (figure 23)
- It even works, if the referenced module is not the same as the current file. (figure 24)

5.3 Linting

Linting is the process of finding errors before compiling the actual thing. Because my server has to parse all the files for tokens and symbols, it already knows a lot about them.

If in the process of parsing, my server finds errors, it will send them to the editor which then can display them as error messages and squiggly lines under where the cause of the error is.

I provide linting of errors for:

- Import statement fails to import the file because it can't be found. (figure 9)
- Cycles in the import graph. (figure 10)
- Hints of missing imports. This is shown instead of unresolved symbol, if a symbol that might be referenced, can be found. (figure 11)
- Unoptimized imports. That means, duplicate imports and imports that are already indirectly imported via another file. (figures 12 and 13)
- Token count mismatch in *defi* and *trefi* statements where the kind of environment expect a different amount of tokens then provided. (figure 14)
- All kinds of unresolved symbols. (figure 15)

The linting error list can easily expanded and can be a helping hand in keeping files clean from errors.

5.4 Goto operation & Import graph

And finally, the editor also supports goto definition operations. The goto operation is useful because with thousands of definitions, you might forget if something you are searching is that what you think it is. So definitions for all symbols can be displayed for example a module (figure 25) or a defi (figure 26).

Inversely, the usages or references to all symbols can be displayed (figure 27).

As a bonus the extension also supports visualizing the import graph of the currently open module (figure 8). The visualization shows normal imports as black edges. Reimports, which are direct edges that could be removed because another import already imports that file, as red edges. Files that can't be found as dotted nodes with red font. And cycles are highlighted via bright green edges.

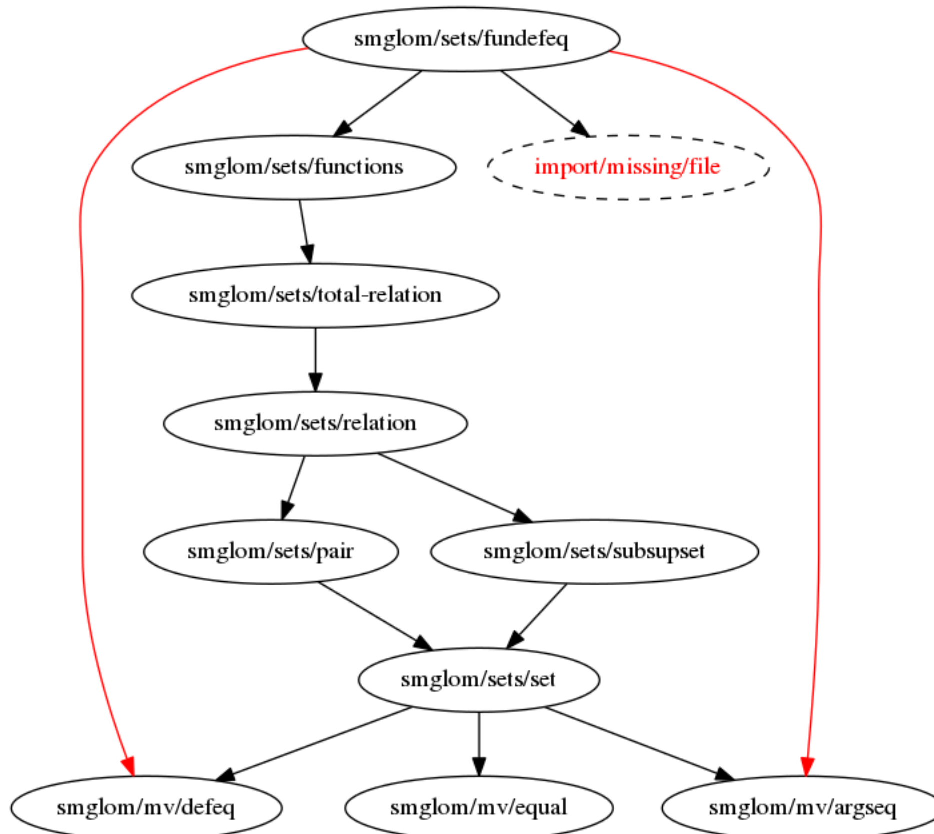

Figure 8: Import graph of a file. Cycles are not in this graph because they make it difficult to understand.

6 Conclusion

I have presented a framework for using a machine learning editor extension to support annotating of keywords in a glossary for mathematical texts.

I showed various machine learning algorithms and evaluated their performances. It appears that a deep model is indeed the best choice currently. And with the implementation of powerful autocompletion and linting tools, I believe that the goal of supporting the human using machine learning was achieved successfully.

But I also would like to do more research using more lightweight models like the hidden markov model, as I did not have enough time to properly implement the algorithm. Luckily, even though deep models are expensive and even with the overhead of the extension, it only takes about 500ms to send the text to the server and receive tags back. At this speed it can still be classified as realtime.

The features from the wikify paper are as well just copied and not adjusted to this problem. I think that it is possible to find better features that incorporate our prior knowledge about the keywords better.

There are also various other small things that could be improved. For example, currently models save their tfidf, keyphraseness, etc. scores with them. But maybe updating them dynamically as the corpus changes might be a good idea. As as it is now, these feature only update by retraining a completely new model with an updated corpus.

The extension needs more work in order to increase stability and fix bugs. There are a lot of small inconveniences that need to be fixed for maximum productivity.

Implementing the heavy part in python and making the extension spawn a server and make use of it's interface, makes it possible to integrate this into almost any editor.

The project with source code for the extension is available at
https://gl.kwarc.info/Marian6814/bachelorarbeit.

7 Images

```
Module "import/missing/module" is missing a module signatu
re file.
\gimport[import/missing]{module}
```

Figure 9: Linting error message for trying to import an import that does not exist

```
\gimport[smglom/primes]{balancedprime}
Module "smglom/sets/fundefeq" reimports "smglom/primes/bal
ancedprime"
• fundefeq.tex(4, 3): Also imported here.
Cycle created by importing module "smglom/primes/balancedp
rime."
• fundefeq.tex(4, 3): Cycle created by importing module
  "smglom/primes/balancedprime."
```

Figure 10: Linting error for cycles in the import graph

```
Missing import module: "smglom/primes/balancedprime"
Unresolved symbol: "balancedprime"
\trefi[balancedprime]{balanced}{prime}
```

Figure 11: Linting error that hints to the module that has to be imported for the symbol

Figure 12: Linting error for duplicate imports in the same file

Figure 13: Linting error indirect duplicate import of a module (reimport)

Figure 14: Linting error for mismatching argument counts

Figure 15: Linting error for unresolved symbols

Figure 16: Displays modules that can be imported without a folder argument.

Figure 17: Displays all reachable folders that contain modules.

Figure 18: Displays modules defined in the specified folder of an import environment.

Figure 19: Autocompletion of a locally defined module inside a mhmodnl environment.

Figure 20: Autocompletion in a trefi argument for imported modules.

Figure 21: Finds names of imported defis after a name= argument of a defi.

Figure 22: Shows names of defis defined after a "module?" trefi argument.

Figure 23: Lists all defi tokens from the local file if no argument is specified.

Figure 24: Lists all defi token texts from the module specified as argument.

Figure 25: Shows the file contents where definition to the module "primenumber" under the cursor is, inline.

Figure 26: Shows the file where the definition to the defi the trefi under the cursor references, inline.

```
\trefii[primenumber]{prime}{number}
m/primes/source – 3 references
balancedprime.de.tex primes/source
    ist eine \mtrefi[primenumber?prime-number]{Primzahl},
balancedprime.en.tex primes/source
    prime} is a \trefii[primenumber]{prime}{number}
primenumber.en.tex primes/source
    number of \trefiis{prime}{number} not greater than $n$ is written as
```

Figure 27: Shows instances where the defi referenced under the cursor is used.

References

[Aba+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[Cho+15] François Chollet et al. *Keras*. https://github.com/fchollet/keras. 2015.

[For73] G. D. Forney. "The viterbi algorithm". In: *Proceedings of the IEEE* 61.3 (1973), pp. 268–278. DOI: 10.1109/PROC.1973.9030.

[Gin+16] Deyan Ginev et al. "The SMGloM Project and System. Towards a Terminology and Ontology for Mathematics". In: *Mathematical Software - ICMS 2016 - 5th International Congress*. Ed. by Gert-Martin Greuel et al. Vol. 9725. LNCS. Springer, 2016. DOI: 10.1007/978-3-319-42432-3. URL: http://kwarc.info/kohlhase/papers/icms16-smglom.pdf.

[Jou+16] Armand Joulin et al. "Bag of Tricks for Efficient Text Classification". In: *arXiv preprint arXiv:1607.01759* (2016).

[JP14] Christopher D. Manning Jeffrey Pennington Richard Socher. *GloVe: Global Vectors for Word Representation*. 2014. URL: http://nlp.stanford.edu/data/glove.6B.zip (visited on 01/16/2019).

[KB14] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[LB02] Edward Loper and Steven Bird. "NLTK: The Natural Language Toolkit". In: *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*. ETMTNLP '02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 63–70. DOI: 10.3115/1118108.1118117. URL: https://doi.org/10.3115/1118108.1118117.

[MC07] Rada Mihalcea and Andras Csomai. "Wikify!: Linking Documents to Encyclopedic Knowledge". In: *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*. CIKM '07. Lisbon, Portugal: ACM, 2007, pp. 233–242. DOI: 10.1145/1321440.1321475. URL: http://doi.acm.org/10.1145/1321440.1321475.

[Mik+18] Tomas Mikolov et al. "Advances in Pre-Training Distributed Word Representations". In: *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*. 2018. URL: https://fasttext.cc/docs/en/english-vectors.html (visited on 01/16/2019).

[Neu] Graham Neubig. *NLP Programming Tutorial 5 - Part of Speech Tagging with Hidden Markov Models*. URL: http://www.phontron.com/slides/nlp-programming-en-04-hmm.pdf (visited on 01/16/2019).

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[PSM14] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: http://www.aclweb.org/anthology/D14-1162.

[Rab17] Ullrich Rabenstein. "Meaning Extraction and Semantic Services in STEM-Documents – A case study on Quantity Expressions and Units". Master's Thesis. Informatik, FAU Erlangen-Nürnberg, 2017. URL: https://gl.kwarc.info/supervision/MSc-archive/blob/master/2017/urabenstein/Rabenstein.pdf.

[Ram03] Juan Ramos. *Using TF-IDF to Determine Word Relevance in Document Queries*. 2003. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.1424&rep=rep1&type=pdf.

[RJ86] L. R. Rabiner and B. H. Juang. "An introduction to hidden Markov models". In: *IEEE ASSp Magazine* (1986).

[Sch16] Jan Frederik Schaefer. "Declaration Spotting in Mathematical Documents". B. Sc. Thesis. Jacobs University Bremen, 2016. URL: https://gl.kwarc.info/supervision/BSc-archive/blob/master/2016/schaefer-frederick.pdf.

[Tou+03] Kristina Toutanova et al. “Feature-rich Part-of-speech Tagging with a Cyclic Dependency Network”. In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*. NAACL ’03. Edmonton, Canada: Association for Computational Linguistics, 2003, pp. 173–180. DOI: 10.3115/1073445.1073478. URL: https://doi.org/10.3115/1073445.1073478.

[Wan16] Alvin Wan. *TexSoup*. https://github.com/alvinwan/TexSoup. 2016.

[Zin04] Claus Zinn. “Understanding Informal Mathematical Discourse”. PhD thesis. Technischen Fakultät der Universität Erlangen-Nürnberg, 2004. URL: https://sites.google.com/site/clauszinn/verifying-informal-proofs/37_04.pdf.