Using Recurrent Neural Networks and Learner Corpus to Detect and Correct Preposition Errors by English Learners

Juan Pablo Garcia Guerrero

Master of Information Technology

Flinders University

May 2018

Supervisor:

Professor David Powers

Submitted to the College of Science and Engineering in partial fulfilment of the requirements for the degree of Master of Information Technology at Flinders University – Adelaide Australia

I, Juan Pablo Garcia Guerrero, certify that this work does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text.

[illegible]

Signed..

Date: 10/05/2018

Abstract

In a global world, English has become a *lingua franca* used for trading, research and general communications whereby millions of people from distinct language backgrounds are learning English as a second or foreign language. Second language acquisition is challenging, and it takes years – on average - before a person can gain English proficiency. The research about this field is plentiful but not conclusive because of the large number of factors affecting the second language acquisition process. Many tools have been developed to support this process within the field of Computer-Assisted Language Learning, but traditional proofreading techniques that are suitable to check native writings are not suitable to check learner writings, whereby recent research around the learner errors detection and correction tasks has emerged with some shared tasks such as Helping Our Own -2011 and 2012 – and CoNLL 2013 and 2014 creating some baselines on which new research can be carried out.

This study undertakes the task of detecting and correcting prepositional errors made by English learners through the use of Recurrent Neural Networks and more in particular Long Short Term Memory architecture. Among the Natural Language Processing techniques, this approach differs from previous works in which it uses non-linear classifiers rather than the traditional linear classifiers such as Naïve Bayes, Maximum Entropy, Average Perceptron, N-Grams and so on. The data used to train, validate and test the algorithm was the National University of Singapore Corpus of Learner English, which is one of the largest Learner Corpora available for research and which is fully error annotated whereby data-driving techniques can be brought about.

Different elements and techniques were proposed, tested and analysed before determining their positive impact on the learner prepositional error correction task and before including them in the main algorithm. On the Recurrent Neural Network side such elements were the attention mechanism, regularization through dropout, the bidirectional model, and deep recurrent neural networks. On the features and examples side, lexical features, part of speech tags, dependency parse indexes, sequence length and number of classes (prepositions) were tested using different values. Moreover, embedding layers – following the skip-gram architecture - were created from the learner corpus to assess their influence on the task at hand by varying parameters such as the window size. The final algorithm, after applying some pre-processing and post-processing techniques, renders promising results when compared to the results of the CoNLL 2013 shared task.

Acknowledgement

I would like to thank my supervisor David Power for his guidance through this project and his wife Sue for her support on the proofreading of this study.

Table of Content

Indices of Tables

Indices of Figures

1 Introduction

As a second language, English is no doubt the first choice around the world (55 counties have English as their second language (MoveHub 2018)) due to its use as a *lingua franca for* trading, research and international communications in general. Millions of people are learning English either in native English speaking countries (Second language learners) or in their own country (Foreign language learners) (Izumi, Uchimoto & Isahara 2004; Ng et al. 2014)[1]. Just in China, reports suggest that over 30 million people are learning English as a second language (Tetreault & Chodorow 2008); and in the United States, around 10% of the population of public schools have English deficiencies as they speak a first language other than English (Chodorow, Tetreault & Han 2007). In 2017, the Department of Education and Training of Australia reported that more than 155 thousand students were enrolled in English Language Intensive Courses for Overseas Students (ELICOS) (Training 2017), which accounts for about 21% of the total of international students. The total number of international students goes up to almost 800 thousand. More than 90% of them speak a first language other than English (Training 2017).

The process of acquiring a second language is complex and some reports suggest that, on average, it can take seven years before a person can gain English proficiency (Hakuta 2011). The development of tools to support this process is then a natural consequence in a world in which technology plays an important role in education. In fact, recent research in error correction is derived from classic automatic proofreading techniques for native English speakers to tackle the task of correcting errors for non-native English speakers (Rozovskaya & Roth 2010b). These traditional techniques cannot be applied successfully to second language learners due to the difference in the language knowledge domain (De Felice & Pulman 2008). Moreover, over the past years, many different learning computing tools have been developed (Lee & Seneff 2006). To what extent and how these tools should be utilized by learners are complex questions studied in the fields of Computer Aided Language Learning (CALL) and beyond the scope of this study. Nonetheless, it is important to highlight that these learning tools have emerged from the complexity of the second language acquisition process and from the clear need of mechanisms to support it.

Any CALL system should be able to identify learning mistakes before tacking any other task, but it is challenging because English grammar is complex and many ambiguities can emerge from a single sentence. In linguistics, six different categories tackle English language ambiguities differently, namely phonology, morphology, syntax, semantics, pragmatics and discourse (Jurafsky & Martin 2009). Syntax and semantics provide different theories and methodologies to detect incorrect grammatical sentences, and they are used to tackle particular types of errors, e.g. wrong spelling, wrong word choice and so on. In the past twenty years, research in the task of detecting and correcting second language (especially English) errors has increased, and the creation of shared tasks tackling the grammatical errors of English learners have created some mileage and evaluation metrics.

In 2011, (Dale & Kilgarriff 2011) proposed the pilot Helping Our Own shared task to detect and correct grammatical errors in order to support the scientific community to improve the English proficiency of researchers who speak languages other than English. Other shared tasks have followed suit, tackling differently the task of detecting and correcting incorrect grammar, i.e. tackling different groups of types of errors. Prepositions and articles are the most common types of errors among English learners (De Felice & Pulman 2008; Leacock et al. 2014). In particular prepositional mistakes tend to be one of

[1] Although within second language acquisition (SLA) there is a difference between second language learners and foreign language learners, for the purpose of this study both terms will be treated as equivalent to each other.

the most challenging types of errors for English learners (Chodorow, Tetreault & Han 2007). In the Cambridge Learner Corpus, it is estimated that 13.5% of written errors made by English Learners are related to prepositions (Leacock et al. 2014). Also, in a study of 53 students between intermediate and advanced level, (Bitchener, Young & Cameron 2005) reported that 29% of the total errors are related to prepositions. In the Japanese learner corpus, (Izumi et al. 2003) reported that 10% of error were prepositional. More significantly, the rate of proportional errors in the available data for the HOO 2011, which was collected from published papers in the Association for Computational Linguistics, was 19% (Dale & Kilgarriff 2011). As a consequence, the goal of this study is primarily the detection and correction of prepositional errors made by non-native English speakers.

The approaches to undertake the task of detecting and correcting prepositional errors vary in type of algorithms and features. Some of the classical approaches involve linear classifiers and language models, but more recently, Neural Networks have been widely used, achieving state-of-the-art Natural Language Processing tasks (Goldberg 2016). Therefore, the prime approach followed in this study to detect and correct prepositional errors is to use Neural Networks, and more precisely Recurrent Neural Networks following a Long Short Term Memory (LSTM) architecture. A wide number of experiments of different recurrent neural network components and features, with their respective analysis, is carried out in this study in order to determine the best parameters and features for the task of detecting and correcting prepositional errors made by English learners. The study makes use of the National University of Singapore Corpus of English Learner (NUCLE), which is one of the largest learner corpora available for research and fully error annotated.

2 Review

2.1 English Learners Grammatical Errors

In the context of identifying categories of grammatical errors made by English Learners, errors vary widely in type and category (Dale & Kilgarriff 2011; Leacock et al. 2014). For example, whereas some lexical errors occur at token level (e.g. character and word) such as spelling errors and word choice errors, other grammatical errors appear in the structure of grammatical forms (e.g. clauses, phrases and sentences) such as sentence structure errors and word order errors. The literature on Learners Error Analysis is wide and rich, and a thorough review can be found in (James 2013). For the context of automatic error correction, however, this study focuses more on empirical error categories that are used to annotate Learner Corpora. The task of fitting the different types of errors into a comprehensive error categorization is a difficult task, and different studies have adopted different sets and subsets of error categories. The errors codes and tagging systems used in Annotated Learner Corpora differ in structure and content; a difference which is deepened by the somewhat subjective interpretations made by examiners and coders about English learners' errors.

One of the first attempts to define a comprehensive error tagging system capable of providing the flexibility that coders need, and the generalization to create statistics of Learner errors was made by (Dagneaux, Denness & Granger 1998). The system was used to tag a subset of the International Corpus of Learner English. They used a descriptive system in which seven linguistic categories ("*Formal", "Grammatical", "Lexico-grammatical", "Lexical", "Register", "Word redundant/word missing/word order" and "Style")* are subcategorized in a hierarchical manner. Overall, their annotating system is a bit complex.

In contrast, in the codification of the Cambridge Learner Corpus, (Nicholls 2003) adopted a more comprehensive system in which five types of errors (*"wrong Form used"* F, *"something Missing"* M, *"word or phrase needs Replacing"* R, *"word or phrase is Unnecessary"* U, and *"word is wrongly Derived"* D) are complemented by a specific word class so that the error can be specified. The system is complemented with punctuation, countability and false friends' errors codes respectively. Finally, 13 additional codes dealing with specific errors such as collocation are added, for a total of 68 possible errors. The system was borrowed and subtly modified by (Dale & Kilgarriff 2011) to define the HOO 2011 shared task, in which the data set consisted of fragments of papers published in the Association for Computational Linguistics (ACL). A much reduced subset of errors was used in the HOO 2012 (Dale, Anisimoff & Narroway 2012), in which only prepositions and determiners were targeted. Concretely, the subset consisted of six types of errors: Wrong word choice (*"Replacement")*, missing word (*"Missing"*) and unwanted word (*"Unwanted"*) for both determiners and prepositions respectively. The corpus used was the publicly available Cambridge ESOL First Certificate in English (FCE).

Another error tagging code was developed by the Centre for English Language Community (CELC) at the National University of Singapore (NUS) and described by (Dahlmeier, Ng & Wu 2013) in their description of the NUCLE corpus. The error tagging system consists of 13 categories ("*Verbs", "Subject-Verb agreement", "Articles/Determiners", "Nouns", "Pronouns", "Word choice", "Sentence structure", "Word order", "Transitions", "Mechanics", "Redundancy", "Citations"* and *"Others"*), which are in turn specified into 27 error categories (e.g. Verb error can be specified as Verb Tense, Verb Modal, Missing Verb and Verb Form errors respectively). A modified subset (*"Article or determiner", "Preposition", "Noun number", "Verb form"* and *"Subject-verb agreement")* of this tagging system was adopted by (Ng et al. 2013) in the definition of the CoNLL 2013 shared task. In the CoNLL 2014 shared task, the number of error categories was extended to 28, i.e. all error tags in the NUCLE corpus.

In general, prepositional mistakes made by English learners can be classified as wrong prepositional choice (e.g. Original: *"we arrived to the station",* Corrected*: "we arrived at the station"*), missing preposition (e.g. Original: *"we are fond Ø beer",* Corrected: *"we are fond of beer"*) and redundant or unwanted preposition (e.g. Original*: "he went to outside",* Corrected: *"he went outside"*) (Tetreault & Chodorow 2008).

2.2 Correcting Learner Errors

Over recent years an increasing interest in error detection and error correction has emerged in the research community thanks to the creation of the shared tasks proposed by Helping Our Own (HOO) (Dale, Anisimoff & Narroway 2012; Dale & Kilgarriff 2011), Computational Natural Language Learning (CoNLL) (Ng et al. 2014; Ng et al. 2013) and Automated Evaluation of Scientific Writing (c) (Daudaravicius et al. 2016)[2]. These shared tasks have created an evaluation basis on which new studies can work so that comparisons between the performance of different approaches can be carried out. Nonetheless, many studies dealing with the task of detecting and correcting errors in learner writings (especially articles and prepositions) had been carried out before these shared tasks appeared (De Felice & Pulman 2008; Gamon et al. 2008; Tetreault & Chodorow 2008).

Although some error correction methods and proofreading tools were basically heuristic and rule-based at the beginning, they are these days data-driven approaches to a large extent (Leacock et al. 2014). This does not mean that rule-based techniques are no longer used (actually some modern error correction systems are supported by rule-based modules), this means that the huge amount of data that is collected day by day allows the construction of simpler systems that are less dependent on linguistic knowledge, i.e. statistical models can encode the set of rules that good grammars follow by looking at good grammatical sentences (likewise, they can encode grammatical error patterns by looking at Learner corpora).

Because errors vary in type, traditional error detection and error correction systems use specific modules to deal with specific errors. For example, in the HOO 2011 shared task, most teams chose to tackle specific errors among the 28 error types defined in the shared task (Dale & Kilgarriff 2011). This tendency was more remarkable in the HOO 2012 shared task in which a reduced number of errors was targeted (determiners and prepositions), so that most teams used two separate modules to deal with determiners and prepositions respectively (Dale, Anisimoff & Narroway 2012). Among the methods to detect and correct learner errors, two types stand out: classifiers and sequential models (e.g. n-grams and language models). All these approaches share a common feature: they are all based on feature engineering. However, recent studies on Neural Network models suggest that error correction tasks can be achieved without feature engineering (Liu & Liu 2017). This study will test if the knowledge domain in grammar, especially on prepositions, can improve the evaluation metrics when applied to NNs or if NN architectures can by themselves discover these features.

2.3 Classifiers

A typical approach to detect and correct written errors is to create modules for each type of target error behaving as classifiers. A multiclass classifier seems a natural approach to deal with errors related to closed classes such as articles and prepositions. Early and recent works on detecting and correcting errors made by learners, have made use of different types of classifiers (Chodorow, Tetreault & Han 2007; De Felice & Pulman 2008; Han, Chodorow & Leacock 2006; Izumi, Uchimoto & Isahara 2004; Tetreault & Chodorow 2008). (Dahlmeier, Ng & Ng 2012), for instance, designed and

[2] However, the AESW shared task is focus on a general estimation of the observation of papers with respect to some guidelines rather than on particular error corrections.

implemented two separate multiclass classifiers to detect and correct article and preposition errors respectively from a finite set of classes in the shared tasks HOO 2012. They defined a confusion set consisting of 36 classes from the 36 most frequently used English prepositions. Among the different classifiers used to detect learners' errors the most common are Maximum Entropy Classifiers, Naïve Bayer Classifiers and Averaged Perceptron Classifiers. Because these classifiers are linear, the hypothesis functions differ from one another in their weights (Rozovskaya & Roth 2011).

2.3.1 Classes

The number of classes (prepositions) to target in an error-detection and error-correction system should, ideally, be equal to the total number of prepositions. However, it is rarely the case. One simple reason could be that the exact number of preposition is not known (see the section on prepositions). However, the real reasons are practical: performance and convenience. On the one hand, the number of prepositions that English learners use is small (Leacock et al. 2014), and on the other hand, evidence suggests that by tackling the most common prepositions, systems can detect most of the prepositional errors. For example, (Rozovskaya, Sammons & Roth 2012) used only ten prepositions for all their system, and although (Dahlmeier, Ng & Ng 2012) use a total of 36 prepositions for their prepositional module, they used only seven prepositions (*"about, at, for, in, of, on, and to"*) in their system to deal with missing prepositions as these cope with most of the occurrences of these type of errors, and they detected that adding other prepositions did not improve their system performance. It may be that small examples for some prepositions are not sufficient to capture their behaviour. This study aims to prove this assumption.

Moreover, because for each prepositional error in a sentence the set of words belonging to the prepositional class are potential correction candidates, Rozovskaya and Roth aimed to narrow the scope of candidate corrections by building a confusing set of prepositions per context. The context was dependent on the first language of the writers, that is, they grouped prepositions in smaller candidate prepositions sharing some grammatical characteristics given by the first language (L1) of the writers, which they called *L1 dependent candidate set*. In addition, on the basis that given a prepositional error, different sets of prepositions may have a different likelihood of occurring, and that prepositions are dependent on the first language of the sentence's author, Rozovskaya and Roth generated sets of confusion arrays for L1. They created two types of confusion sets: 1) a confusion set of all ten prepositions, and a confusion set for a particular misplaced preposition *pi* consisting of the prepositions that correct *pi*, that is to say, of all prepositions that are confusable for an English Learner with a particular L1 background. Although this approach is inviting, in this study no subsets will be created because of two main reason: time constrains and the lack of data.

2.3.2 Features

An important step in a classification task is the selection of features, from which classifiers should be able to capture enough domain information during training so that new instances can be successfully solved in production (Dahlmeier, Ng & Ng 2012). However, feature engineering is an art that has not been mastered yet. For example, (Dahlmeier, Ng & Ng 2012) chose a group of features for their prepositional error-detection and error-correction system based on empirical experiments. In general, lexical features together with Part-Of-Speech (POS) tags, N-grams and chunks are commonly used for the task of detecting and correcting grammatical errors (Dahlmeier, Ng & Ng 2012).

Lexical features are important in *context-sensitive* tasks such as correcting spelling errors (Jurafsky & Martin 2009), and correcting prepositional errors (Rozovskaya, Sammons & Roth 2012). In these scenarios, it is common to use N-gram windows of words around the target without including it. (Rozovskaya, Sammons & Roth 2012) chose as features for their system to detect and correct prepositions n-grams windows going from one to four on both sides of the target preposition, e.g. one word before, two words before, one word after, two words after and so on. They also include the head of the prepositional phrase. (Dahlmeier, Ng & Ng 2012) used lexical features (words) around the observed preposition and their respective POS tags. Their novelty in respect to previous works was the introduction of the observed preposition among the features. The values of the features were binary, i.e. the feature appears in the example (value 1), or it does not (value 0). (Tetreault & Chodorow 2008) also introduced in their prepositional system a set of lexical and grammatical features comprising a window of words around the preposition (two before and two after) and their respective Part-of-speech tags. They also included the head of the preceding grammatical components, i.e. noun phrase, verb phrase as well as the head of the following components; for a total of 25 features.

2.3.3 Naïve Bayes

Naïve Bayer Classifiers are probabilistic and generative classifiers, i.e. they generate the likelihood of a class based on the prior probability of the data observations by applying the Bayes theorem (Manning, CD, Raghavan & Schütze 2008). The algorithm is a highly cost-efficient method that is suitable to use when there is a computational limitation (e.g. memory and CPU) or a limitation in the training data (Vasilis 2015). Although it is a very simple model in many cases, its performance can be as good as more complex approaches. In terms of detecting and correcting English Learners' errors, (Rozovskaya & Roth 2011) proved that, although it is outperformed by other methods such as Maximum Entropy and Language Models, it suits models in which the first language of the English learners is taken into account very well. In fact, (Rozovskaya et al. 2013) used a Naïve Bayes Classifier to detect and correct prepositional errors in the CoNLL 2013 shared task. However, despite the fact that their system outperformed all other systems when the five types of errors defined by the shared task were computed, they were outperformed in the prepositional errors by the statistical machine translation (SMT) model of (Yoshimoto et al. 2013).

2.3.4 Maximum Entropy

Maximum Entropy ME (Jaynes 1957) Classifiers fall into the class of probabilistic models such as the well-known Naïve Bayer Classifiers and N-Grams. However, unlike them, the ME classifier is a discriminative classifier, i.e. the model is constructed over the conditional probability of the hidden classes (e.g. prepositions) given the observations (e.g. the extracted features) (Manning, C 2005). ME classifiers are a good choice for multiclass classification when the prior distributions of the data are not known well enough. In training, the feature weights that best maximize the conditional likelihood of the data are chosen by applying the Principle of Maximum Entropy (e.g. the feature weights that obtain the maximum entropy value among all feasible models) (Manning, C 2005). Maximum Entropy models start by setting a uniform probability for all classes, and then some feature constraints, fitting the data set more precisely, are progressively added. Finally, applying the principle of Maximum Entropy, the best model (feature weights) will satisfy the feature constraints of the training data, and it will *"assume a distribution of maximum entropy"* to new (unknown) data (Chodorow, Tetreault & Han 2007).

In one of the first attempts to apply NLP methods to L2 error correction, (Izumi, Uchimoto & Isahara 2004) used a ME classifier to detect and correct L2 writing errors (Although they used a Japanese Speech Corpus). They considered that the error correction tasks are similar to the tasks of text categorization, in which ME classifiers have been largely used. The features that they used were basically contextual (lexical), and for the tasks of detecting and correcting omission errors and replacement/insertion errors, they claimed a precision of 75.7% and 31.17% and a recall of 45.67% and 8% respectively. Their system targets a total of 13 types of errors (including prepositional errors) and they did not present results for individual types of error so it is difficult to judge their system performance on prepositions. (Han, Chodorow & Leacock 2006) also implemented a ME classifier but it aimed to detect and correct determiner errors only.

On the other hand, (Chodorow, Tetreault & Han 2007) used an ME classifier to tackle the detection and correction of prepositions that were wrongly selected, i.e. wrong choice of preposition. They chose ME because evidence suggests that it works well on heterogeneous features, and without the need of assuming independency between the features as in a Naïve Bayer. Their system was complemented with some rule-based filters, showing that prepositional classes have some similarities that make them confusable. This also shows that new approaches are required to capture the slight difference between some prepositions. They claimed a precision of 80% and a recall of 30%. (Tetreault & Chodorow 2008) took a step further with their model to achieve a precision of 84% and a recall of 19%, using 25 different lexical and POS features around the target preposition. They added a number of filters to maximize precision (affecting recall), i.e. to minimize the chance of marking a proposition as wrong when the probability between different options is slight, and therefore, there exists a chance of wrong correction or of various prepositions fitting the context but with some semantic vitiation that is hard to verify.

2.3.5 Average Perceptron

The Averaged Perceptron AP ("*Voted Perceptron*") is a discriminative algorithm developed by (Freund & Schapire 1999), which takes the Classical (*"vanilla)"* Perceptron algorithm of (Rosenblatt 1958) a step further by making use of online learning algorithms. The main novelty of the AP with respect to the vanilla perceptron is that it stores (or remembers) the weights of all features so that later training examples do not have greater counts than early training examples (Daumé III 2012). Additionally, AP is a mistake-driven algorithm, i.e. weights are updated only when the classifier with the current feature weights classifies wrongly on the training data (Rozovskaya & Roth 2011). The AP classifier works for linear classification tasks, and works more efficiently in training than other algorithms (e.g. Support Vector Machine SVM and Logistic Regression) on data that can be linearly separated (Freund & Schapire 1999).

In an empirical study, (Rozovskaya & Roth 2011) show that when trained on the same data, the Averaged Perceptron outperforms the Naïve Bayer classifier and other statistical models such as Language Models and the *"counting"* model SumLM. In addition, they found that it could be up to twice as efficient as a Naïve Bayer Classifier. It is not surprising that for the HOO 2011 shared task, (Rozovskaya et al. 2011) used a regularized version of the Averaged Perceptron to detect and correct articles and prepositions. For prepositions, they achieved the best score with an F-score of 0.488, 0.488 and 0.363 for tasks of detection, recognition and correction respectively. The university of Illinois (Rozovskaya, Sammons & Roth 2012) trained an AP classifier once more to tackle prepositional tasks in the HOO 2012 shared task, in which their system scored first in the tasks of detection and recognition, while they were second in the task of revision. However, as they trained the system on the learner corpus FCE dataset in which, even for learners, the density of prepositional errors is sparse, they found that the AP is *"sensitive"* to the density of errors. So, they introduced a method ("error inflation") to artificially increase the number of errors in the dataset according to their respective proportion.

2.3.6 Multiclass classifier using the Confidence-Weighted Learning algorithm

In the domain of NLP tasks, in which features are high-dimensional, online learning algorithms seem a more natural option than typical batch learning algorithms because they are updated on the basis of a single example at a time. However, the occurrence of some feature instances is rare but potentially essential for discovering the best possible weights for the task of classification. As a consequence, (Dredze, Crammer & Pereira 2008) introduced the confidence-weighted (CW) algorithm, in which parameters are updated in an inverse proportion to confidence. The level of confidence is measured using a probabilistic model for each feature, which is defined by a Gaussian distribution, and updated after each new instance is processed during training. In other words, unlike traditional learning algorithms that use a single vector of weights, the algorithm uses weights that follow a multivariate Normal distribution defined by a mean μ and a covariance matrix €. The learning algorithm iterates over a pair of labelled data (x_i, y_i) at a time, all throughout the data set (Y, X) (Dahlmeier, Ng & Ng 2012). The algorithm was used by (Dahlmeier, Ng & Ng 2012) in the HOO 2012 to train the different classifiers they used in their system, because the algorithm proved suitable for sparse and high dimensional features. They system achieved the highest F-score (28.7) for the task of correcting prepositions.

2.3.7 Binary Classifier

So far, all analysed classifiers have been used as multiclass classifiers, but another option is to use binary classifiers. They are a good choice if the problem is analysed from the perspective of omission, i.e. whether a specific preposition should or should not be placed at a certain position. (Dahlmeier, Ng & Ng 2012) trained ten binary classifiers, one for each preposition, for their missing prepositions module. For each classifier the confusion set was composed of the preposition and the absence of the preposition (*"empty preposition"*). They took examples from every noun phrase where a preposition preceding it was a positive example and the absence of a preposition was a negative example (or when the wrong preposition was preceding the noun phrase). They followed a similar approach to tackle unwanted prepositions with the only difference being that, in training, positive examples were prepositions marked as unwanted, whereas prepositions correctly used were negative examples.

2.4 Language Models

It was said that the construction of a rule-based system to detect and correct prepositional errors made by non-native English speakers requires linguistic expertise and high grammatical knowledge. Data-driven classifiers, on the other hand, although more flexible, require the extraction of features capable of rendering enough information to model and therefore generalize the behaviour of the target goal. These features can be either shallow or deep (although a combination of them is more common), and Language Models usually require surface-based features that depend solely on the count of feature occurrences (Elghafari, Meurers & Wunsch 2010). This characteristic of Language Models make them quite handy and in some cases, as will be seen bellow, they can outperform classifiers with complex features.

2.4.1 N-grams

Although the early *"simple Markov chains"* (bi-grams) of (Markov 1913) and the N-Grams of (Shannon 1948) were popular in the 50's to model sequences of words, the works of (Chomsky 1956), stating that the Markov models were *"incapable of being a complete cognitive model of human grammatical knowledge"* (Jurafsky & Martin 2009), drove away linguistic and computational research from statistical models. Nonetheless, they are today a popular method to model languages thanks to the works in speech recognition of (Bahl, Jelinek & Mercer 1990; Baker 1975; Jelinek, Bahl & Mercer 1975). N-Grams are statistical models of sequences of symbols (characters, words or sentences), in which a Markov assumption is applied, i.e. the probability of the N symbol (say word) is assumed to depend only on the N-1 preceding words (Jurafsky & Martin 2009). In practice, the conditional probability of each symbol given the N-1 preceding words is calculated by counting the actual number of occurrences of the sequence in a given corpus. Because some sequences cannot be present in the training corpus but can occur in production, many smoothing techniques are applied. (Jurafsky & Martin 2009) gives a good account of these techniques.

(Elghafari, Meurers & Wunsch 2010) carried out an empirical study to find out how much predictive information the lexical words (shallow features) surrounding a preposition can contain in order to detect and correct prepositional errors. They used a 7-Gram including the preposition, and they followed a web-as-corpus approach to maximize the size of the training set, i.e. they constructed a cohort of nine prepositions (they used the top 9 prepositions according to the BNC corpus) in their training set (BNC corpus) that was sent to the search engine Yahoo. Using the full back-off approach of (Katz 1987), they reached an accuracy of 76.5%, proving that using only surface-features could be as efficient as using more complex features. (Boyd & Meurers 2011) followed the same approach for 4 different types of errors (conjunctions, determiners, prepositions and quantifiers) in the HOO 2011 shared task, but instead of using a web-as-corpus approach for the task of counting, they used the ACL anthology. Overall, their system performed the best when detecting substitution errors of prepositions and determiners as they detected 67% of errors.

2.4.2 Statistical Machine Translations

Statistical Machine Translation (SMT) models take a string of symbols (e.g. words and phrases) as an input, and translate them into an aligned target language using statistical language models. The alignment process is not trivial because the resulting string can include *"reorderings, omissions, insertions, and word-to-phrase alignments"* (Och & Ney 2003). In general, the SMT model can be defined as the subset of the Cartesian product of the source strings and the target strings. However, to make the general representation feasible, a number of restrictions are applied to the resulting string so that the number of possible outcomes depends on the source string and its context (Och & Ney 2003). The best translation is the model that, parameterized by a specific vector of weights, achieves the highest probability given some set of features. The size or level of the string source could go from word-based (Och & Ney 2003) to phrase-based (Koehn, Och & Marcu 2003) (it can even be applied at a character level, e.g. spelling correction).

In the context of detecting and correcting English learners' errors, (Brockett, Dolan & Gamon 2006) carried out some experiments on correcting noun mass errors using a SMT as they detected that learners' errors were not isolated but were a sum of factors such as context and other errors that could be treated as a language (bad English) that could be translated into another language (good English). (Mizumoto et al. 2012) scoped "*all*" types of errors using the phrase-based model of (Koehn, Och & Marcu 2003), and they found that, when correcting English leaners' errors, local context provides enough information to translate from grammatically incorrect sentences to grammatically correct sentences. For prepositional errors, more precisely, (Mizumoto et al. 2012) showed that the SMT could outperform a ME classifier. In fact, (Yoshimoto et al. 2013) ranked first in the task of detecting and correcting prepositional errors in the CoNLL 2013 shared task with an F-score of 17.53 using a similar approach to (Mizumoto et al. 2012), but training the SMT model on a larger corpus.

More recently, (Felice et al. 2014), in the CoNLL 2014 shared task, used a SMT in their hybrid system to generate the top ten candidate corrections for all (28) types of errors, which were later filtered by some Language Models to choose the ultimate correction. Overall, their systems perform the best when compared to the gold-standard edits with an F05 of 37.33, and it obtained the highest recall (38.26) for prepositional errors.

2.5 Prepositions

Prepositions are hard to master by English learners because, unlike some grammatical rules about the syntax of English, they do not have specific rules for their usage. That is to say, while it is clearly stated that noun phrases cannot follow intransitive verbs, there is not a clear rule defining which preposition is the best choice to use, given certain contexts (Leacock et al. 2014). Moreover, because this study has a strong focus on prepositions, it is worthwhile to first understand what prepositions are, how they behave and which features they are characterised by.

2.5.1 What are prepositions?

Prepositions are an open class of words which are estimated to contain more than a hundred tokens (Cambdridge), but whose exact amount varies on different reports as they are interpreted differently by distinct authors, and because they vary as time goes by (Huddleston & Pullum 2005; Jurafsky & Martin 2009). In his book, Lindstromberg (Lindstromberg 2010), for instances, analyses only 90 English prepositions that he considered were of common use today and he decided not to examine archaic prepositions such as *betwixt* and *outwith,* or prepositions coming from the Latin and which are not commonly use like *cum* and *circa*. On the other hand, Huddleston and Pullum (Huddleston & Pullum 2005) include in the class of prepositions some other words that commonly are grouped into either adverbs or subordinating conjunctions.

Words belonging to the class of prepositions normally precede a noun phrase (Jurafsky & Martin 2009), and by grammatical rule, the noun phrase should be a complement of the preposition (Huddleston & Pullum 2005). For example, in one of the examples given by Huddleston and Pullum, the word *since* can be classified as preposition in the sentence *"I haven't seen her, since Easter*", whereas it falls into the class of conjunctions in the sentence *"I haven't seen her since she left town",* because the component that follows "*since"* is a clause and not a noun phrase. Other rules state that prepositions do not inflect nor are gradable (as are adjectives and adverbs) and, frequently, they are used to represent some kind of relation (e.g. a relation of time or space) (Huddleston & Pullum 2005; Jurafsky & Martin 2009). Some examples taken from Jurafsky and Martin are: a literal relation <u>*on*</u> *it* and a metaphorical relation <u>*on*</u> *time* (Jurafsky & Martin 2009). The semantic relation can appear in arguments to point out either the indirect object of an action (*"He gave a book <u>to</u> Mary"*), the author of an act in a passive form ("*the book was written <u>by</u> Fred*"), the instrument with which an action was carried out on ("*they ate the cake <u>with</u> a fork*") or the source from which something comes *("Jane took the vase <u>from</u> the shelf"*) (Leacock et al. 2014).

While describing the class of prepositions, Huddleston and Pullum (Huddleston & Pullum 2005) compare prepositions with adjectives, adverbs and verbs from which the following properties and examples were extracted: 1) Prepositions can be the head of a prepositional phrase that can be placed at different positions in a sentence, and which usually depend on either a noun (*"a <u>house</u> <u>at</u> the beach"*) or a verb (*"He <u>saw</u> her <u>at</u> school."*). 2) When prepositions appear at the head of an adjunct, they do not have to be associated with the predicand (*"<u>After</u> the end of the semester, the dean threw a party for the students", "<u>After</u> the end of the semester, there was a party for the students."*). 3) Prepositions, acting as predicative complement, can be complement of the verb *be* but they are less frequently of other verbs such as *feel, appear, seem* and they are never complement of the verb "*become"* ("*We are <u>in</u> your debt."*, but not "*We became <u>in</u> your debt."*). 4) Some verbs in their gerund-participle or past participle form can take the functionality of prepositions, but they can be distinguished from the preposition function because, as predicator, the verb has to be associated to a subject, whereas as a prepositional phrase does not (As preposition: *"<u>Following</u> the meeting, there will be a reception."*, as verb: *"<u>Following</u> the manual, <u>we</u> tried to figure out how to assemble the unit.").*

Among the class of prepositions, a smaller group composed basically of the words *"as, at, by, for, from, in, of, on, than, to, and with"* (Huddleston & Pullum 2005), can have a grammatical function in which they act as a mark within the structure of a sentence that is not dependent on the meaning of the preposition, i.e. the preposition could not be substituted without having to alter the grammatical structure of the sentence. Huddleston and Pullum called these prepositions “grammaticised prepositions”, and some of their examples are depicted next. For example, in the sentence *“I sat by the door”,* the preposition *by* has a spatial meaning that could be replaced by *opposite* (*“I sat opposite the door”*), without altering the grammatical structure. On the other hand, the function of the preposition *by* in the sentence *“The article was written by a first-year student”,* is to mark the subject of the clause in the participle form and cannot be substituted by other prepositions without having to modify the grammatical structure of the sentence. Some of these grammatical functions of the prepositions are to mark the subject of a sentence in the passive form (*“The article was written by a first-year student”*), to mark the subject within a noun phrase in a sentence (*“The sudden death of the president stunned the nation.”*), to mark the complements of a verb *(“I transferred several hundred dollars to them.”),* to mark the complement of a noun (*“Their request for assistance was ignored.”*) and to mark the complement of an adjective (*“They all seem quite keen on the idea.”*) (Huddleston & Pullum 2005).

2.5.1.1 Collocations

Collocations are pairs (or groups) of words that have a strong association with each other and whose form (sequence of words) creates a particular meaning, i.e. if a word in the phrase were replaced with another one with similar meaning and usage, the resulting phrase would be incorrect or odd. Lindstromberg (Lindstromberg 2010), for instance, depicts how the preposition *in* in the collocation *in trouble* could not be replaced by *inside,* despite the fact that *in* and *inside* share a similar usage. Some collocations allow the insertion of a word (e.g. an adjective) to strengthen the expression such as *in big trouble*, but it is not possible in all collocations. For example (again from Lindstromberg), adding the adjective “*extreme”* between the collocation “*at random”* would render a weird phrase: “*at extreme random”* (which is sintactically correct). This type of collocation belongs to the category of fixed collocations, that is, collocations that do not permit alterations.

2.6 Training Data

In the domain of error correction, three different types of training data are commonly used for training machine learning algorithms: 1) native English corpora, 2) artificial learner corpora (made from native corpora) and 3) learner English corpora (Leacock et al. 2014). Since the Brown Corpus (Francis & Kucera 1964), the first online English corpus (Jurafsky & Martin 2009), appeared, the number of the first type of corpora has increased with a relatively good availability for research purposes. A relevant corpus that is largely used for NLTK is the Penn Treebank (Marcus, Marcinkiewicz & Santorini 1993), which can be used through the Natural Language Toolkit (Bird, Klein & Loper 2009). Some approaches to tackle the error correction task were carried out using this type of corpus. (De Felice & Pulman 2008), for instance, trained a Maximum Entropy Classifier to correct prepositional errors using the British National Corpus (BNC). However, the particular patterns of learners’ grammar is not present in these type of corpora and the learning algorithms do not capture the English domain knowledge of L2, limiting thus the potential to detect and correct grammatical mistakes. In fact, (De Felice & Pulman 2008) presented results for their classifier on L2 that were slightly less accurate than results on L1. As a consequence, many researchers opt for populating native English corpora with learners’ errors following some available distribution of learner errors. (Rozovskaya & Roth 2010c) introduced learner errors into English Wikipedia following a learner error distribution so that the patterns seen during training could be as close as possible to the patterns seen during testing, i.e. they generated an annotated learner English corpus from a native English Corpus. (Rozovskaya, Sammons & Roth 2012) followed this method to populate the Google Web 1T corpus (Brants & Franz 2006) with the proportion of errors found in the FCE dataset for their prepositional system in the HOO 2012 shared task. These type of methods were a consequence of the low availability of annotated learner corpora.

The number and size of annotated learner corpora has been to some extent limited. One interesting project is the International Corpus of Learner English (ICLE) (Granger et al. 2009), which in its second version provides English writings from 16 different L1s. Unfortunately, this rich diversity of first languages cannot be exploited for error correction tasks as the corpus is not annotated, leaving this corpus for the sole use of linguistics[3]. The same lack of annotated data applies for the Chinese Learner English Corpus (Gui & Yang 2003). Some small annotations were carried out by (Rozovskaya & Roth 2010a), but unfortunately the size does not reach a desirable level. On the other hand, the Cambridge Learner Corpus (CLC) (Nicholls 2003) may be the most desirable learner corpus for research with a unique size of over 16 million words written by learners from 86 different first languages other than English. It is estimated that more than 6 million words have been annotated. Unfortunately, the whole corpus is not completely available for research purposes, with the relatively small FCE Dataset made available by the Cambridge University Press (Yannakoudakis, Briscoe & Medlock 2011). Nonetheless, it is an interesting corpus that was already used in the HOO 2012 shared task. An even larger and available learner corpus is the NUS Corpus of Learner English (NUCLE) (Dahlmeier, Ng & Wu 2013), with over one million words. The corpus was used in both CoNLL 2013 and CoNLL 2014 shared tasks. Although this corpus lacks the diversity of the FCE Dataset, it is a learner corpus with the size and features to work as a baseline for automated error correcting tasks, and it is the main corpus used in this study.

2.7 Neural Networks Models

As we saw in the chapter of classifiers, most of the models described in the literature to correct learner errors are linear classifiers, which use sparse vectors as inputs. This makes perfect sense as they are the basis upon which many NLP techniques work. However, neural networks have been used recently to tackle NLP tasks successfully, achieving state-of-the-art results (Goldberg 2016; Sutskever, Vinyals & Le 2014; Vinyals et al. 2015), so it is worth exploring the possibilities that neural networks open for error correction tasks and that seem not to have been exploited yet. (Liu & Liu 2017) used neural networks (LSTM architecture) for detecting grammatical errors for the purpose of discriminating correct scientific language from improper scientific language, which was the main purpose of the shared task Automated Evaluation of Scientific Writing Shared Task 2016 (Daudaravicius et al. 2016). They claimed that their model outperformed a support vector machine architecture as well as another popular neural network architecture: convolutional NN. However, although this task is somehow related to the task of error correction, the ultimate goal is completely different and it is worth exploring the solution of the error correction task with neural networks. Another interesting novelty of neural networks is the possibility of introducing dense vectors into non-linear classifiers (Mikolov, Tomas, Chen, et al. 2013).

Historically, Artificial Neural Networks (NNs) were based on the human brain. The first model was the *"Threshold model"* by (McCulloch & Pitts 1943) in which the authors made several assumptions about neural networks that led them to treat their model as a binary device that uses a threshold of constant value. Their model calculated the dot product between a vector input and the weights of a neural network to carry out *OR* and *AND* logical operations. (Hodgkin & Huxley 1952) developed the more complex *"spiking model"* which treated each component of a squid's giant neuron as electrical components. Other complex interpretations and models have been developed (Izhikevich 2003; McGregor 1987) through the years. However, more simplistic representations such as the perceptron (Rosenblatt 1958) provided more practical implementation of the brain representation with promising results (Goldberg 2016). However, in 1969, (Minsky & Papert 1969) published a number of criticisms

[3] In fact, the corpus was acquired to carry out this study, but the lack of annotated data prompted the use of another corpus: NUCLE.

regarding the limitations of neural networks that created a sense of prejudice and frustration against neural networks, negatively affecting the funding for neural networks research. Despite this limitation, important discoveries were brought about by a small group of researchers such as the backward propagation algorithm (Werbos, P 1974) and the Cognitron (Fukushima, Miyake & Ito 1983). Moreover, some advances during the 70s and 80s attracted the attention of the scientific community once again, prompting in this way a re-emerging of the neural network field (Siganos & Stergiou 1996).

These days, two general neural network architectures are popular, namely recurrent networks and feed-forward networks (Goldberg 2016). Feed-forward networks are non-linear classifiers that can be easily adapted from linear classifiers to tackle NLP tasks. (Chen & Manning 2014), for instance, trained a feed-forward neural network to train a dependency parser classifier. Their model used a hidden layer and an embedding layer (detailed in a next section) which was added to the input. They reported great improvement in speed performance as well as an improvement in the accuracy of the parser. Similar approaches were carried out by (Weiss et al. 2015) and (Pei, Ge & Chang 2015) with similar results. (Vaswani et al. 2013), on the other hand, applied neural networks to machine translation, obtaining an improvement in translation quality across different pairs of languages.

Moreover, convolutional neural networks – which fall into the category of feed-forward networks – can be used for other tasks in which the main information can be found in "*strong local clues*" within an example (Goldberg 2016). (dos Santos & Gatti 2014) used a convolutional neural network for sentiment analysis in short texts such as tweets, achieving an accuracy of 86.4 % in the Stanford Twitter Sentiment corpus (STS), thanks to the capability of convolutional networks to exploit text analysis at character level and to generalize the results to the word level. It is thus not unreasonable to think of using convolutional architectures for the error correction task, because the clue to identify, for example, the right prepositional choice for its dependent component (e.g. say a noun phrase), could be near or far from the prepositional phrase. However, there is another neural network architecture that is even more tempting to use for the error correction task, namely recurrent neural networks.

The main strength of recurrent neural networks (Elman 1990) is that unlike any other classifier – either linear or non-linear - they can capture the structural information inherent in sentences (Goldberg 2016), i.e. while other classifiers inject the features into the neural network in a way in which the order is not relevant, recurrent neural networks maintain the order of the features so that the grammatical structure is injected into the neural network. Unlike typical neural network architectures in which all features are injected at the same time along the input layer, recurrent networks input a single feature (or a concatenation of features such as word, POS, dependency head, etc.) at a time in a sequential way $(x_1, x_2, \ldots, x_n)$, returning a sequence of outputs $(y_1, y_2, \ldots, y_n)$, each output at a time. Recurrent networks keep a state s_i that together with the input x_{i+1}feeds the neural network, and that together with the output y_{i+1} represents the state of the neural network after processing all inputs from x_1 to $x_i$in a sequential fashion. The representational power of recurrent networks is given then by the prediction of an output y_i given all the sequences of inputs $x_1, x_2, \ldots, x_i$ and the states s_0, $s_0, \ldots$ s_{i-1} in a similar way as n-grams but without the limitations of Markov assumptions (Goldberg 2016). The recurrent network architecture implemented in this study will be discussed in more detail in the method chapter.

2.7.1 LSTM

Recurrent networks work naturally with sequences of features and they provide state-of-the-art results for sequence tasks such as Combinatory Categorial Grammar (CCG) super-tagging (Xu, Auli & Clark 2015) and language models (Jozefowicz et al. 2016; Mikolov, Tomáš et al. 2010). That is because given a sequence of features - say words - of arbitrary length, a recurrent network inputs each feature at a time t up to the last feature x_n in order to make a prediction y_n. That is to say, any number of features could be processed using the same recurrent network and, therefore, the same set of parameters θ. The setting of the parameters is critical and it can be achieved using the Backpropagation Through Time (BPTT) algorithm (Werbos, PJ 1990), in which all input timesteps are unrolled to accumulate the errors and then roll back to update the parameters. However, as proved by (Pascanu, Mikolov & Bengio 2013), recurrent networks are hard to train because their gradient becomes too small (almost zero) when the backpropagation algorithm reaches the first inputs of the unrolled network. As a consequence, working with long sequences becomes impractical as distant dependencies are not captured by the recurrent network.

Fortunately, (Hochreiter & Schmidhuber 1997) introduced a new variant of recurrent networks using a memory state (*"memory cells"*) to avoid the vanishing of the gradient, namely the LSTM architecture (figure 1) (although their model did not include the forget gates). The memory cells keep a state of the network through time and they can be accessed through mathematical functions working as logical functions (*"gating components")* that decide how much information is going to be kept in the memory state and how much is going to be discarded (forgotten). Mathematically, a LSTM is defined as:

$$y_j = R_{LSTM}(s_{j-1}, x_j) = [c_j;\ h_j]$$

where c_j represents the memory state and h_j represents the hidden state (output per each timestep).

Figure 1 LSTM diagram (reproduced from (Olah 2015))

Mathematically:

$$c_j = c_{j-1} \odot \mathrm{f} + \check{c} \odot \mathrm{i}$$

$$h_j = \tanh(c_j) \odot \mathrm{o}$$

The symbol $\odot$ is used to represent a pointwise product between vectors. The functions $i, f\ and\ o$ represent the gates that decide what values of the cell state are going to be updated (figure 3), what information is going to be discarded or forgotten from the cell state (figure 2) and what information from the cell state is going to be outputted (Figure 4). The gate $\check{c}$ defines what values are candidate to be included in the cell state (figure 3) and, together with i, defines what new information is going to be added in the cell state. Mathematically, they are defined as:

$$i = \sigma(x_j W^{xi} + h_{j-1} W^{hi})$$

$$f = \sigma(x_j W^{xf} + h_{j-1} W^{hf})$$

$$o = \sigma(x_j W^{xo} + h_{j-1} W^{ho})$$

$$\check{c} = \tanh(x_j W^{x\check{c}} + h_{j-1} W^{h\check{c}})$$

Figure 2 Forget gate (reproduced from (Olah 2015))

One advantage of recurrent neural networks is that, because they can handle sequences in a natural way, there is no need to use techniques to combine variable numbers of features (say the words of a sentence) into a fixed number of features. That is to say, because traditional classifiers and typical feed-forward neural networks require a fixed number of features as input, techniques such as Continuous Bag of Words (CBOW) (Mikolov, Tomas, Chen, et al. 2013) are used to combine (or average) a variable number of features into a fixed number of features:

$$CBOW\ (f_1 \dots f_k) = \frac{1}{k}\sum_{i=1}^{k} v(f_i)$$

On the other hand, RNNs can handle variable numbers of features because they work over time, i.e. one feature is injected at a time t along an infinite – in theory – sequence of discrete features. In addition, as mentioned previously, recurrent neural networks process features in a sequential manner so that the structural information of the group of features is kept, unlike typical bag of words approaches (Jurafsky & Martin 2009). As a consequence, it is not necessary either to engineer features representing the distance and position of the core features (Goldberg 2016), or to embed them into a dense space (Zeng et al. 2014).

Figure 3 input and candidate gates (reproduced from (Olah 2015))

2.7.1.1 Bidirectional Recurrent Network

A more complex elaboration of a recurrent network is the bidirectional model introduced by (Schuster & Paliwal 1997). Recurrent neural network typically work with sequence of features belonging to the past, i.e. previous words. The bidirectional model introduces an extra neural network that keep the state of features belonging to the future, i.e. words that come after the target. That is to say, a bidirectional model keeps two separate cell states (and two RR nodes) working in opposite directions: while one node works in a forward way keeping a memory of the past features, a second node works in a backwards way keeping the state of the future features. The final output is given by the concatenation of both recurrent networks. This model has been applied successfully in NLP tasks such as sequence tagging (Irsoy & Cardie 2014) in which the tag at timestep t can be determined by both the preceding tags as well as the following tags. This approach seems promising for the task of correcting errors and therefore it is tested in this study.

Figure 4 output gate (reproduced from (Olah 2015))

2.7.1.2 Deep Recurrent Networks

One of the limitations of artificial intelligence (AI) in the past was the fact that some problems, that were possible to formulate formally, could be solved in quite a straightforward way by applying traditional computing and in a bit more complex way by applying artificial intelligence algorithms. However, AI's relevance was evident when tasks that were easy to solve for humans such as image and spoken processing were hard to define formally, and become too complex to solve by applying traditional computing (Goodfellow et al. 2016). Machine learning algorithms learn from experience -

the examples (data) that are used to feed them - to understand the knowledge domain of the task at hand. Internally, they build a *"hierarchy of concepts"* about the knowledge domain that developers do not have to formulate explicitly (Goodfellow et al. 2016). Theoretically, shallow neural networks (say MLP1) have the representational power to approximate a large set of functions, i.e. they are a *"universal approximator"* (Hornik, Stinchcombe & White 1989), but in practice it has been seen that the capability of neural networks to learn the knowledge domain from experience depends on the number of layers (the depth of the network) the network is built with (Goldberg 2016). That is to say, the hierarchy of concepts created by each layer becomes more abstract and has a bigger representational power as the number of layers increases, and that is why this concept is known as Deep Learning (Goodfellow et al. 2016).

Nonetheless, because of their nature, the depth of recurrent networks is hard to define and deep recurrent networks could be thought of as long neural network nodes unrolled over a long range of time. However, the function of the internal layers of recurrent networks is to keep a memory state of the sequence rather than to build a hierarchy of processing layers, which is the inherent purpose of neural networks and, additionally, recurrent networks output a result for each input so that there is no extra processing for each timestep (Hermans & Schrauwen 2013). To overcome this potential deficiency, some researchers (Graves, Mohamed & Hinton 2013; Hermans & Schrauwen 2013) have successfully developed recurrent neural networks with multiple layers that they call stack of recurrent networks, and that is based on the early work by (El Hihi & Bengio 1996). The representational power that can be gained using deep recurrent neural networks is desirable for the task of error correction because many layers could embed the algorithm with a bigger representational power and, therefore, deep recurrent networks will be tested in this study.

2.7.2 Additional LSTM Neural Network Components

In recent literature, many different components to improve the accuracy of neural networks and machine learning techniques in general have emerged with promising results such as Dropout, Attention and Embedding Layers. In theory, all these techniques would be a desirable component in a neural network where there is a limited amount of training data, and for the task at hand it is worthwhile testing if the integration of these components improves the performance of the error correction system.

2.7.2.1 Dropout

Overfitting – overtraining the data and undergeneralizing the domain problem - is an issue that affects deep neural networks and that can be addressed somehow through regularization methods such as L_2 Regularization, in which a squared penalty is applied to each parameter, but which seems to work better for logistic methods (Goldberg 2016). A more recent alternative for neural networks – especially when applied to feed-forward neural networks – is the Dropout method, proposed by (Srivastava 2013). The idea behind this method is to drop some neurons during training so that the network does not learn to rely on just specific weights. The way in which the dropout is applied to the network seems to have an important impact on its effectiveness (Zaremba, Sutskever & Vinyals 2014). Nonetheless, some promising reports suggest that dropout improves the performance of neural networks (Pham et al. 2014; Zaremba, Sutskever & Vinyals 2014) and it is tested in this study.

2.7.2.2 Attention

Although internally a LSTM neural network is designed to keep a memory cell, for some NLP tasks such as machine translation, it has been detected that the encoding and decoding process is affected by the burden of keeping a memory state of all sequence of words in a fixed-size vector. As a consequence, the implementation of an extra memory cell helps the recurrent neural network to learn to distinguish the relevant sections of the sequence, i.e. the RNN learns to pay attention to the key features within the whole sequence of features (Bahdanau, Cho & Bengio 2014). This technique is known as an attention mechanism and has proved to be effective for image processing (Denil et al. 2012; Larochelle & Hinton 2010) and NLP tasks (Bahdanau, Cho & Bengio 2014; Liu & Liu 2017). The introduction of an attention mechanism makes sense in the context of preposition correction because not all words in a phrase help to determine the more suitable preposition. Therefore, this mechanism is tested in this study.

2.7.3 Feature Embedding Layer

These days, a popular technique that, together with Neural Networks, is used to tackle NLP tasks is the embedding of high dimensional inputs into fixed n-dimensional vectors that are called dense vectors (Goldberg 2016). It was first proposed by the work of (Bengio et al. 2003) in which they tried to generalize the idea of learning the joint probability of word sequences that are used in traditional n-grams concatenations. Their model aimed to *"learn a distributed representation of words"* so that both a statistical representation of words and a model to generate the probability of word sequence were obtained. The concept was applied by (Collobert & Weston 2008) to deal with multiple NLP tasks in which they transformed the word indices of a dictionary into a vector, and the vector representation was accessed through a look-up table. Moreover, (Chen & Manning 2014) applied the embedding concept to different type of features, e.g. words, tags etc.

Two prime advantages are expected to be obtained from the use of dense vectors: first of all, there should be a significant computational gain as the large number of features used in NLP tasks, say words, produce huge one-hot representation vectors. Traditional NLP techniques represent lexical features as one hot vectors, i.e. each word is represented as a huge vector of zeros whose size is equal to the size of the vocabulary, and in which the value of the index of the word feature is one. However, the real gain is expected to be the sharing of semantic meaning between features in an n-dimensional space. According to (Mikolov, Tomas, Chen, et al. 2013), when large amounts of data are available, many simple techniques can outperform more complex systems that use smaller training data sets. However, as it has been discussed, the availability of large learner corpora is scarce or not available for research (e.g. the Cambridge Learner Corpus). Therefore, the use of dense vectors that share statistical information is a desirable feature to be included in the model to correct learner errors.

The semantic sharing of features using embedding layers is given by the fact that while in a one-hot representation the sequence of features "the car is good" is as different as both "the van is good" and "the house is good", in a dense representation, it is expected that the features "car" and "van" are closer than the features "car" and "house" in the n-dimensional representational space. Therefore, in a dense representation, features with similar semantic content are related to each other in an n-dimensional space, giving thus, in theory, an extra statistical strength to the neural network input and a bigger capability to generalize that should increase the accuracy of the algorithm during testing (Chen & Manning 2014; Mikolov, Tomas, Chen, et al. 2013).

2.7.3.1 Word2vec

One of the most successful and widespread implementations of an embedding vector for features is the word2vec approach proposed by (Mikolov, Tomas, Chen, et al. 2013). They proposed two architectures using a log-linear classifier to achieve their embedding system, namely Continuous Bag of Words (CBOW) and skip-gram. The first architecture consists of an input that is projected into a shared layer so that all words (or features) are *"averaged"* over the same position. The name *"Bag of Words"* comes traditionally from the fact that the sequence of words is not taken into account in this architecture; but the prefix *"continuous"* indicates that, unlike traditional approaches, it uses continuous representations of the context, i.e. it predicts the current feature depending on the continuous history of the context. On the other hand, although similar to CBOW, the skip-gram model aims to predict a feature depending on another feature of the same local context (e.g. sentence) (Mikolov, Tomas, Chen, et al. 2013). That is to say, each current feature is injected into the classifier as an input to predict features in a window of features after and before the current feature. Although the computational complexity increases with the size of the window, it is expected to also increase the quality of the final embedding. However, a weighting system should be included in the model because if the distance between the current feature and the target increases, the effect or dependency of the target over the current feature decreases (Mikolov, Tomas, Chen, et al. 2013).

According to the optimized skip-gram model presented by (Mikolov, Tomas, Sutskever, et al. 2013), each pair of valid examples of the form (word, target) is classified in a binary way against a randomly created pair of samples (noisy examples) of the form (word, random target) in order to teach the algorithm to discriminate between real target words and noisy ones (i.e. negative sampling). Formally, the goal of the model is, given a set of words ($w_1, \dots, w_T,$) surrounding the target word and a training context of size c, *"to maximize the average log probability":*

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log \mathrm{p}(w_{t+j}|w_t)$$

The skip-gram applies a softmax function over the context words to obtain the probability distribution of the word features:

$$p(w_O\,|w_I) = \frac{e^{(v'^{T}_{wO}|v_{wI})}}{\sum_{w=1}^{W} e^{(v'^{T}_{w}|v_{wI})}}$$

Where W is the size of the vocabulary and v_w and v'_w are the input and output of the *"vector representation of w"* (Mikolov, Tomas, Sutskever, et al. 2013). In practice, applying softmax over such a large number of elements (W words) is too expensive and the authors proposed the hierarchical softmax architecture introduced by (Morin & Bengio 2005) (in the domain of neural networks language models) in order to reduce the computing cost of applying the softmax function to generate the probability distribution of the embedding features. In the hierarchical softmax, each word is a leaf in a tree representation in which each node represents the relative probability of all words that can climb up to the node, i.e. all child nodes. However, (Mikolov, Tomas, Sutskever, et al. 2013) found a better alternative using the Noise Contrastive Estimation (NCE) by (Gutmann & Hyvärinen 2012) in which the differentiation of data (real values from noisy ones) is achieved using a logistic regression. Mathematically, using the NCE, each target in the skip-gram model $logP(w_O\,|w_I)$ is replaced by the term:

$$\log\sigma({v'_{WO}}^{T} v_{W1}) + \sum_{i=1}^{k} E_{w_i\sim P_n(w)}\left[\log\sigma(-{v'_{WO}}^{T} v_{W1})\right]$$

The feature embedding layers can render beneficial effects on the task at hand as the relatively small size of the learner corpus could not provide enough lexical examples per type. As a consequence, the semantic sharing could alleviate this weakness and embedding layers are tested in this study.

3 Method

From the promising results of neural networks on NLP tasks and the relative success of the use of classifiers in previous works to correct learner errors, the main model tested in this study is a neural network classifier that aims to predict n different classes of prepositions given some features out of a context. For the task of error correction, the algorithm picks up the class that best fits a given context and proposes it as the right preposition to be used for that context. If the proposed preposition differs from the original preposition, then the algorithm flags an error and proposes the predicted preposition as the correct preposition. Nonetheless, because the context can be ambiguous and the confidence of the proposed correction low, two further analyses have to be taken into account before making a decision. First, it needs to be determined during the analysis of the results how well the algorithm can predict a particular class. For this purpose two types of graphs are presented in the results chapter per experiment and per class: the Receiver operating characteristic (ROC) graph and the Precision-Recall graph (further detailed in the results section). Second, the appropriate threshold (or confidence) at which the algorithm should achieve the best metrics has to be determined as well. A new graph in which all metrics are drawn against a threshold is also presented in the results chapter. All this information is used to perform a post-processing whereby the confidence of the algorithm should increase in order to reduce false predictions.

The core of the model is a recurrent neural network that follows a Long Short Term Memory (LSTM) architecture. The use of a LSTM architecture as the core of the algorithm was an expected choice because it deals in a natural way with sequences (Sutskever, Vinyals & Le 2014; Vinyals et al. 2015) and, in the task of correcting learner errors, the context of the task is given by a sequence of features, i.e. words, POS tags, etc. Moreover, following related works, new promising features were progressively added to test the feasibility of improving the performance of the algorithm over the task at hand by adding these new features. The results were carefully analysed to isolate confusing variables that could affect the judgement over the influence of the new feature or component on the output of the algorithm. In the coming sections, each experiment affecting the general model will be explained in more detail.

In order to progressively run and test all the different experiments (components and features), a flexible solution was implemented[4] using the programming language Python (Python 2018). The system controls the behaviour of the algorithm through a number of defined arguments whereby the system controls the deployment of the core algorithm according to the experiment under study. The system provides default values so that only a few parameters should be included when running a particular experiment. All arguments are clearly explained in each section. Moreover, in order to implement, train and test the recurrent neural network, the library TensorFlow (Abadi et al. 2016) was used. TensorFlow is an open source machine learning library that provides implementation for different neural networks architectures such as LSTM. Other libraries such as numpy, lxml, nltk and autocorrect were used to support the implementation of the system and its internal configurations. The algorithm was trained using the Google ML-Engine platform.

A final experiment in which the *"optimum"* algorithm according to the results obtained from all other experiments is run over the test data used in the shared task CoNLL 2013 (Ng et al. 2013) in order to compare the prepositional error correction task obtained by the proposed algorithm against a public reference. This "optimum" algorithm is complemented by a post-processing task that consists of making decisions (predictions) based on a threshold that is defined for class and which is obtained

[4] The source code can be obtained from https://github.com/garc0062/prepositional_error_correction.

from the analysis of the precision-recall graphs as well as the ROC graphs. The final algorithm is called in this study the final algorithm.

3.1 Evaluating the Algorithm

Metrics to evaluate the effectiveness of an algorithm to correct learner errors differ in the literature. The two most common ways to measure the effectiveness of these algorithms are Recall and Precision (Leacock et al. 2014). One could include Accuracy as well, but it tends to give a wrong impression (too positive), of the real effectiveness of an error correction algorithm because, even for English learners, the rate of grammatical errors is low and the number of true negatives is high (De Felice & Pulman 2008). To be able to calculate all these metrics, the results obtained from the algorithm must be categorized into four categories, namely true positives (TPs), false positives (FPs), true negatives (TNs) and false negatives (FNs) (Powers 2012).

A contingency table, in which the performance of the algorithm is validated against the true (correct) values, is created for this purpose. The table works for a binary classification where, on the one hand, two classes such as wrong prepositional choice and correct propositional choice are defined as positive and negative examples respectively. In the task of error correction, one can define errors as positive examples, i.e. the example contains an error (e.g. a wrong prepositional choice). On the other hand, correct prepositional choice examples account as negative examples, i.e. the example does not contain any error (at least not the error the classifier is dealing with). In the contingency table, the number of positive and negative examples of the class are distributed between the results of the predictor (table 1), thus generating the four categories (Powers 2012).

	Positive class	**Negative class**
Positive prediction	True positives	False positives
Negative prediction	False Negatives	True negatives

Table 1 Contingency table

In error correction, a true positive occurs when the system flags a true error and the gold standard correction matches the system correction. On the other hand, if the error does not exists it counts as a false positive. Likewise, a true negative occurs when the system does not flag any error and in fact no error exists (or it is not tagged in the gold standard). False negatives are, then, instances in which the system does not flag any error but an error does exist in the gold standard correction.

From these values it is possible to calculate Precision (equation 1) and Recall (equation 2). From the instances tagged as positives by the system, precision measures in which proportion a classifier identifies true positive instances. On the other hand, recall measures what proportion of true instances (errors) are actually flagged as errors. From precision and recall it is possible to calculate F-scores such as F1 (equation 4) and F05 (equation 5), which calculate a harmonic mean of both precision and recall. Moreover, it is also possible to calculate the accuracy of a classifier (equation 3), but as was stated previously it gives a too positive impression of the actual effectiveness of a classifier because the low rate of errors results in a large number of true negatives (Leacock et al. 2014). Nonetheless, precision, recall and F-scores can be affected by the skewness of the data, meaning that all these metrics are not portable from one corpus to another, which makes it difficult to compare the performance of one algorithm against another (Powers 2012). Ultimately, all evaluation metrics have their strengthens and weaknesses (Chodorow et al. 2012). Fortunately, the different shared tasks on error correction described previously have given a baseline on which to compare error correction algorithms, and that is another important reason to use the NUCLE corpus in this study. Additionally, these metrics are the common measures that are used in the literature and if any comparison with previous results is intended, it makes sense to keep the same metrics.

$$Precision = \frac{TPs}{TPs + FPs} \quad (1)$$

$$Recall = \frac{TPs}{TPs + FNs} \quad (2)$$

$$Accuracy = \frac{TPs + TNs}{TPs + FNs + FPs + TNs} \quad (3)$$

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (4)$$

$$F_{0.5} = \frac{(1 + 0.5^2) * Recall * Precision}{Recall + 0.5^2 * Precision} \quad (5)$$

Automatic evaluation of error correction systems requires an annotated corpus in which the output of the predictor (i.e. the classifier) can be compared against a gold standard set of corrections. In this respect, the availability of an annotated corpus such as NUCLE makes the evaluation task easier and more unbiased as one can rely on the annotated data as a gold standard that one can then validate against the output of the error correction algorithm. Nonetheless, error correction is a hard and non-standardized task because different evaluators can come up with different corrections for the same error, even if they intend to annotate the same correction (Dahlmeier & Ng 2012).

(Dahlmeier, Ng & Wu 2013), for example, carried out a pilot experiment to validate the level of agreement between correctors. They used three different criteria, namely Identification, Classification and Exact. In the first criterion two correctors agree if they identify an error within the same span. They also agree on classification if they identify the same type or class of error. The ultimate agreement occurs when they both select the same correction too. Dahlmeier, Ng and Wu used the Cohen's Kappa coefficient (Cohen 1960) to assess the agreement between the annotators. The Cohen's Kappa coefficient is calculated by obtaining the probability of agreement (Pr(a)) and the probability of chance agreement (Pr(e)). The coefficient values that the authors obtained on average were 0.3877, 0.5484, and 0.4838 for identification, classification and exact correction respectively. According to the range of metrics defined by (Landis & Koch 1977), their identification agreement was *"fair"* (from 0.21 to 0.40) and their scores for classification and exact correction were *"moderate"* (from 0.41 to 0.60). Nonetheless, an annotated learner corpus is an immensely beneficial asset for the task of training and evaluating an error correction system.

Similarly, another issue that has to be dealt with when evaluating an error correction system is to interpret true positives. As stated above, a true positive occurs when the system detects an error that was also tagged in the gold standard annotations. However, one can differentiate two types of true positives: 1) the system and the gold standard flag an error in the same span, i.e. error detection. And 2) the system suggests a correction that matches the correction annotated in the gold standard[5], i.e. error correction. Moreover, as seen above, valid corrections may disagree, generating false positives that should be true positives. To smooth this discrepancy, several approaches have been proposed. (Dale & Kilgarriff 2011), for instance, grouped evaluation metrics into three categories, namely detection (an error is detected), recognition (the type and span of the error is identified) and correction (the correction proposed matches the gold standard). Because only prepositional errors are dealt with in this study, just the detection and correction metrics are adopted. For the detection

[5] Bear in mind that a true correction that does not match the gold standard counts as a false positive.

evaluation, the identification of an error by the algorithm regardless of the correction is enough to count as a true positive. On the other hand, for the evaluation of the correction task, true positives count only if the correction suggested by the system matches with the gold standard. It can be visualized in the WAS evaluation Schema defined by (Chodorow et al. 2012), in which an extra line (*) is given to differentiate detection tasks from correction tasks (table 2). If the system, the gold standard, and the predictor differ in the preposition then it is a true positive for the error detection task whereas it is both a false positive and a false negative for the error correction task. The WAS evaluation schema was used in this study to generate the contingency table.

	Written	**Annotated**	**System**
TN	X	X	X
FP	X	X	Y
FN	X	Y	X
TP	X	Y	Y
*****	X	Y	Z

Table 2 WAS evaluation Schema

Moreover, the evaluation metrics proposed in the shared task CoNLL 2013 (Ng et al. 2013) are also taken into account to assess the algorithms proposed in this study. It consists of sets of edits that are compared against each other. On the one hand, there is the set of gold standard edits and, on the other hand, there is the set of edits suggested by the error correction system. The performance is then measured according to the number of edits that intersect both sets over the number of edits proposed in both sets, i.e. precision is calculated as the number of elements of the intersection subset of both sets over the total of elements of the system edits set. On the other hand, recall is calculated as the number of elements of the intersection subset of both sets over the total of elements of the gold standard edits.

3.2 The Data

The NUCLE corpus used in this study was the pre-processed version 2.0 made by the shared task CoNLL 2013. They present the data as a sequence of lines per token in the NUCLE corpus with a lot of information, namely the essay ID, the paragraph index, the sentence index, the token index, the token itself, the POS tag, the index of the parent in the dependency tree, the dependency relation with the parent node and the constituent parse tree in a bracket-like format. All this additional information was obtained through the Stanford parser (Klein & Manning 2003) and the Python library NLTK (Bird, Klein & Loper 2009). The data extracted from this version differs slightly from the NUCLE version 2.3 presented by (Ng et al. 2013) as shown in table 3.

	NUCLE 2.0	**NUCLE 2.3**
Essays	1397	1397
Sentences	57,146	57,151
Tokens	1,161,604	1,161,567
Word types	29352	-
Prepositions	133093	-
Prepositional Errors	1341	-
Prepositional Errors Rate	1.01%	-

Table 3 NUCLE data Overview

Order	**Preposition**	**Number of instances**	**Prepositional Errors**	**Rate**
1	Of	32236	193	0.60%

2	In	20085	242	1.20%
3	For	10704	215	2.01%
4	To	9645	167	1.73%
5	As	7867	11	0.14%
6	That	7328	1	0.01%
7	On	7021	162	2.30%
8	From	5934	34	0.57%
9	With	5706	66	1.16%
10	By	4884	47	0.96%
11	At	2132	54	2.53%
12	If	1849	0	0%
13	Than	1394	0	0%
14	Into	1232	45	3.65%
15	About	1124	24	2.13%
16	Because	1008	0	0%
17	Like	924	0	0%
18	Since	849	0	0%
19	After	778	0	0%
20	Through	750	18	2.4%
21	Over	676	11	1.62%
22	During	654	12	1.83%
23	Without	630	0	0%
24	Although	610	0	0%
25	While	588	0	0%
26	Whether	468	0	0%
27	Before	436	0	0%
28	Besides	417	1	0.24%
29	Under	410	0	0%
30	Between	404	4	1.00%
The rest	-	4350	34	0.78%

Table 4 Top 30 of Prepositions in NUCLE

3.2.1 Defining Prepositional Errors in the NUCLE corpus.

As discussed previously, the type of prepositional errors a learner may make can be classified into three categories: extraneous insertion of preposition (unwanted preposition was included), missing preposition (no necessary preposition was included) and wrong prepositional choice (preposition used is not adequate or is incorrect). For the NUCLE corpus, these errors can be identified as follows:

- Insertion error: correction indexes include a preposition in original text and, correction tag does not include any preposition – the correction may be empty as well.
- Missing error: correction indexes does not include a preposition in original text and, correction tag does include a preposition.
- Wrong choice: correction indexes include a preposition in original text and, correction tag includes a preposition as well. Usually, it is a one to one correction. Otherwise it could include other type of errors, and thus only one to one examples are taken into account.

However, this study deals only with prepositional errors of the type wrong choice. The same algorithm can be applied to the other two types of prepositional errors but the examples have to be generated differently. It is important to note that because the prepositional class is not well defined (as discussed in the prepositions section), some words that some people could argue are conjunctions or adverbs such as ‘that’ and ‘as’ are treated as prepositions in this study (they are also tagged as prepositions in the NUCLE corpus provided by the shared task CoNLL 2013).

3.2.2 Training, validating and testing datasets

Depending on the type of experiment, the number of examples per class varies, but as a general rule one example is generated for each prepositional instance in the corpus. Furthermore, the examples generated are randomly separated into two parts: 10% of the total data is employed for the testing phase and the remaining 90% is employed for the training phase. The dataset for the training set is randomized and separated into two blocks once again: 90% of the data is used to train the model while the remaining data is used for validating the data through cross-validation. These distinct sets of data are labelled according to their purpose (training, testing and validation) and then saved as Python *“pickle”* objects so that the same datasets can be employed in different experiments. The reason for taking this measure is to avoid that different datasets could bias the judgement over the results, because the data is randomly separated each time the script runs and different datasets are created each time. However, if the system detects that the datasets for that particular type of example already exists, the datasets are loaded instead of being created.

3.2.3 Oversampling and Undersampling

According to the pre-processed data provided by CoNLL 2013, the total number of prepositions contained in the NUCLE corpus is 133093[6] and, as can be seen in table 4, the number of examples for each type of class is quite imbalanced. Just comparing the top three prepositions, it is easily identifiable that the number of instances of the class *“in”* is twice the size of the class *“for”*, which in turn is three times smaller than the top class *“of”*. As a consequence, some methods are applied to the training data to assess if either oversampling or undersampling are a better option for the task of error correction. It is necessary to point out that neither the validation dataset nor the testing dataset are pre-processed, i.e. they retain the imbalanced nature that would be expected in any learner corpus – and probably in real life.

In order to test the undersampling of the training dataset, the parameter *“--under”* is set at True and an extra parameter – *“—under_pos”* – is used to set the maximum number of examples each class can have. The number goes from the top preposition (e.g. *“of”*) up to the bottom one, i.e. a value of 1 means the top preposition, a value of 2 the second one and so on. To set a value that goes below the tenth preposition does not make sense as the number of examples becomes too low, affecting negatively the performance of the neural network. As a consequence, the undersampling is tested by setting the position parameter at 2 and 3 - which are the ones with the largest number of instances - and then 7 and 10 - which are reference points of a significant decrement in the number of examples. The system internally calls a function that takes the reference preposition and set its number of appearances as the reference limit over which classes with more instances are truncated in a random way. The effectiveness of the undersampling method is established by the analysis of the different metrics.

The oversampling, on the other hand, is activated by setting the parameter *“—over”* at True and then setting the minimum of examples each class should have in a similar way to the undersampling parameter, i.e. the parameter *“--over_pos”* defines the position from top to bottom of the preposition that serves as a reference for the minimum number of examples for each class. The oversampling is carried out by applying a bootstrapping-like method, i.e. classes with low numbers of occurrences are

[6] It is important to note that the preposition *“to”* is always tagged as the POS tag “TO”, regardless of its function as either preposition or auxiliary for the infinite verb form, and an extra pre-processing was necessary for this specific class of preposition.

randomly picked up during the training phase in order to compensate for the lack of examples. The parameters are set at values of 2, 3 and 7 per experiment. The effectiveness of the oversampling is tested in a similar way as the undersampling experiment. The default value for both oversampling and undersampling is False thereby all experiments are run using the training data as it is.

3.3 The Model

The algorithm is a multi-class classifier consisting of a LSTM neural network that is fed by a sequence of features (type of examples, e.g. sequence of words), and whose final state, in turn, feeds a logistic regression algorithm. Each node of the output of the logistic represents how well a class of preposition fits the current context or input, i.e. the preposition with the highest score is the one that best fits the current context. In the next sections each component is described in more detail. The neural network is trained by applying the Adam optimization algorithm (Kingma & Ba 2014) to the softmax cross entropy loss function with logits. Both functions are implemented in TensorFlow, and the algorithm calls the functions *"tf.train.AdamOptimizer"* and *"tf.nn.softmax_cross_entropy_with_logits"* respectively.

3.3.1 LSTM Neural Network

A recurrent network that follows a LSTM architecture is the core of the algorithm used in this study to detect and correct learner errors because, in theory, this model can render better results than those presented by linear classifiers. For the task of error correction, LSTM networks can be employed as a statistical machine translation in which, given a sequence of learner English words, it returns a sequence of correct English words through a process of encoding and decoding (Sutskever, Vinyals & Le 2014). Although promising results have been obtained in other fields following this approach, it has the disadvantage – from the CALL point of view – that it does not highlight the specific place in which the error is made, which one would think is a requirement by traditional CALL so that learners can understand their mistakes. Language model is another approach that can be accomplished by using LSTM networks. Given a sequence of words it predicts what the most likely word is that should follow in the sequence or – in the prepositional context - which preposition should be the more appropriate choice. This approach bears some similarity with a classifier approach, but classifiers have one advantage over a language model: in a classifier previous words are not the only context but words after the target preposition are also taken into account. This is especially relevant in the case of prepositions because the right preposition could depend on the argument –usually a noun phrase - it is the head of. Therefore, the approach that this study follows is a classifier that tries to predict the right preposition given a surrounding context.

3.3.1.1 Simple LSTM

A LSTM network can be defined by two components: the size of its state cell and the number of layers in a stack of LSTM nodes. These parameters represent the simplest LSTM model that can be built in TensorFlow. The system that was developed in this study has two components: on the one hand, it controls the type of model that can be trained and tested on the system and, on the other hand, the type of examples that are used by the model to feed the core model during training and testing – the goal of the testing is always the same though. The library TensorFlow provides a number of implementations of neural network models and the package "tf.contrib.rnn" is especially useful for the creation of recurrent cells. Simple LSTM networks were developed by calling the function *"LSTMCell"* that provides an implementation for a LSTM network. Additionally, the systems makes use of the function *"nn.dynamic_rnn"* – which is especially helpful for training, that is, a TensorFlow implementation of an unrolled recurrent network. In order to facilitate the undertaking of different experiments, the system includes a number of parameters that control the type of model and the type of data that should be deployed. The parameter that defines what type of model to use in the system is *"—model_type"*. For a simple LSTM model, the value is *"simple"*. The model is tested against the bidirectional model that is detailed in the next section.

3.3.1.2 Bidirectional

In addition to the simple LSTM network model, that keeps a memory state of all previous features, there exists the bidirectional model – the biLSTM for the scope of this study – which keeps the state of both previous features and future features by means of two separate LSTM networks – a forward LSTM network and a backward LSTM network - and their respective memory cells. For the task at hand, which is similar to the task of tagging, the bidirectional model seems to be a more suitable model as the preposition target is usually located somewhere in the middle of two related sequences of words. Because the bidirectional model consists inherently of two LSTM networks, the model was built on the same base as the simple LSTM model with the difference that the two LSTM networks - the forward network and the backward network – were wrapped by a different unrolling implementation, namely the *"bidirectional_dynamic_rnn"* that controls the state of the two different networks. The value of the type parameter for this model is *"bi"* and this model is compared with the simple LSTM were all other parameter keep their default values.

3.3.2 Dropout

It makes sense to think that building deep recurrent networks with a large number of layers and a large number of hidden cells to correct prepositional errors can lead the RNN to overfit the relatively small training data. Therefore, it makes sense to implement a mechanism to stop overfitting by using a regularization method such as dropout. The implementation of dropout in TensorFlow is relatively straightforward as it provides a wrapper to add dropout to each LSTM cell. Following the recipe by (Zaremba, Sutskever & Vinyals 2014), the dropout is only applied to the *"non-recurrent"* nodes within the network in order to best employ regularization through dropout, i.e. dropout is applied only to the LSTM cells that are located between the cell that receives the input and the cell that outputs the final state. Two parameters are added into the system to control the dropout behaviour: *"--dropout"* which is a Boolean variable that determines if the system must or must not wrap each LSTM cell with a dropout mechanism; and *"--keep_prob",* which indicates the proportion at which neurons should be kept, i.e. a value of 1 indicates that all neurons should be kept. In order to test the effectiveness of adding dropout to the neural network, a relatively deep neural network was set at a number of layers of 8 and a hidden size of 1600, which is the equivalent to the larger parameters defined in the neural network size section. Additionally, the probability of keeping neurons was set initially at 0.5 and progressively increased by 0.1 up to 1.0.

3.3.3 Attention

In theory, the attention mechanism should be appropriate to use when the sequence of features is long, but as it cannot be clearly established from which value the length of the sequence starts to be long, the attention mechanism was tested using the larger size of the input sequence, i.e. for a maximum length of 20 features on each side of the target preposition. The implementation of the attention mechanism in TensorFlow is not very different from the implementation of the dropout wrapper, and a wrapper provided by TensorFlow is used to add the attention mechanism to each LSTM node when the parameter *"--attention"* is set at True. The neural network using the attention mechanism was compared to another neural network of equal size and type that does not used the attention mechanism.

3.3.4 Classes

After deciding what type of algorithm will be used to predict prepositions given a context, it is now time to decide which types of classes the algorithm will address. Ideally, all prepositions should be addressed but two main problems arise: first, the number of preposition is not well defined (see prepositions chapter) and as a consequence different English corpus correctors might annotate a word as a preposition while other might annotate the same word as a conjunction or as any other class – which would affect the supervised training of the model; second, the availability of the data is scarce and the number of examples for non-popular prepositions is minimum and, therefore, the algorithm could not learn properly to distinguish the context that triggers those propositions. Thus, the number of prepositions is reduced to the most popular ones (at least while the model is trained solely on learner data). However, it is necessary to determine what exact number of prepositions should be appropriate to deal with the prepositions that are more difficult to master by learners, but that could be generalized by the algorithm taking into account that the dataset is relatively small (in the Data section it could be seen that for non-popular prepositions the number of instances is small).

To define this number, a parameter was introduced in the code that sets the number of prepositions the algorithm addresses (*"—class_limit"*) so that a default algorithm could be trained using different numbers of classes, i.e. different types of prepositions. The number of classes was set at a minimum of 5 classes and then progressively increased by 5 up to a maximum of 30. The experiments were tested using the testing dataset. In addition to the metrics defined in the section on evaluating the algorithm, two types of graphs were created for each experiment: precision-recall graphs and ROC graphs. The purpose of these graphs was to evaluate for each experiment, how the algorithm was capable of distinguishing each class in a one-vs.-all approach in order to define what prepositions the algorithm should address.

3.4 Size of Neural Network

It is not enough to define the core of the algorithm, it is also necessary to define the size of the neural network. LSTM networks have a cell memory that keeps a state of the neural network through time, and the LSTM network can be parameterized according to the size of this cell state that affects the capability of the network to keep a robust state of the full sequence of features. Additionally, LSTM neural networks can be parameterized by the depth of its architecture, i.e. the number of layers in the stack of LSTM networks. It might be expected that a bigger cell state would render better results as the neural network would be able to retain more information such as long-range dependencies through time, but at the same time it is true that a bigger network would imply a larger use of memory and more computational processing in both training and testing phrases – and production of course. Similarly, a larger number of layers would, in theory, render a more powerful representational capability according to deep learning theory and consequently deeper recurrent networks require more machine resources than shallow ones. The representational power of deep neural networks comes from the learning capability of deep hierarchical layers to capture the key information from the input features by abstracting the domain knowledge in a hierarchical manner. As a consequence, the predictor is better informed, i.e. more precise.

However, the ideal size of neither the cell state nor the number of layers can be determined theoretically. It has to be decided through a series of experiments to make a trade-off between cost and benefits. The way in which it was carried out in this study was by varying the size of each parameter while all other parameters in the algorithm kept their default values. The parameters of the size of the recurrent network were separately tested, that is, each parameter was modified and tested while the other parameters kept a fixed value. The first parameter to be tested was the number of hidden layers, i.e. the size of the cell state. For this experiment, the stack of LSTM cells was composed of only one layer. The basic number of layers was set at 50, and then doubled each time, that is to say, 100, 200, 400, 800 and 1600 per experiment. Next, the number of layers was set at a fixed value (400) and the number of layers was doubled each time up to a value of 8: 1, 2, 4 and 8. The system defines two input parameters to parameterize the number of layers and the number of hidden cells, namely *"—num_layer"* and *"—num_hidden"* respectively.

3.5 Feature Embedding Layer

According to the literature on embedding vectors, two advantages can be gained from its use: a computational gain and an improvement on the representational power of the neural network. Moreover, among the models that are proposed, the context-predicting ones – concretely the skip-gram model - seem to render better results than the count-based embedding vectors (Baroni, Dinu & Kruszewski 2014). As a consequence the word2vec model proposed by (Mikolov, Tomas, Sutskever, et al. 2013) and implemented by (Steiner 2015) – and which is available on the TensorFlow web-page tutorials - is the embedding model followed by this study. Two parameters are important in this implementation – in addition to the examples: the length of the windows at each size of the target word and the number of times each word is used to generate a label – the method by (Mikolov, Tomas, Sutskever, et al. 2013) requires undersampling to get a more powerful representation. Therefore, those two parameters can be tuned during deployment thanks to the introduction of two parameters in the system, to know, *"—size_window"* and *"—skip_num"*. However, the *"—skip_num"* is set at 1 so as to get the biggest representational power of the embedding layer. Another parameter that is relevant in the embedding layer is the size (dimension) of the embedding vector, and in the system this parameter is controlled through the argument *"--word_emb_size"* (at least for lexical features).

In order to evaluate the effectiveness of the embedding layer, a pilot experiment was carried out in which a simple LSTM neural network was trained using each type of input (embedding vectors and one hot vectors) in order to compare both the training time and the evaluation metrics and graphs. The parameter used to control the type of input is *"—type_input"*, which can take the values *"one_hot"* or *"embedding"*. Nonetheless, as the parameters of the embedding layer could affect the performance of the neural network, the embedding layer was further tested by varying the two parameters that control the embedding process, i.e. the size of the window and the size of the embedding vector. To avoid confusing results, each parameter was tested by varying one parameter while the other parameter kept a constant value per experiment. The window size parameter was increased by 1 from 1 to 4, whereas the embedding size parameter was tested at values of 50, 100 and 200. The data used to create the embedding table is equivalent to the whole NUCLE corpus for each type of feature.

Although the name "*word2vec*" suggests that this technique is designed to embed lexical features, other types of features such as POS are tested using both types of inputs. It is important to point out that different embedding vectors can be created from each core feature depending on their function, e.g. an embedding vector for previous word and another one for next words (Goldberg 2016). However, because of the nature of RNNs in which all features are treated the same, in this study all features of the same type will share the same vector. That is to say, because RNNs take care of the position of each feature, and all features have the same equivalence at its respective injection timestep t, it makes sense to use the same embedding vector for all features sharing the same characteristics, e.g. an embedding vector for words and another one for POS and so forth. Nonetheless, because different types of features have different embedding vectors, but they are injected into the neural network at the same time, e.g. lexical features and POS features, all resulting embedding vectors per feature are concatenated through a concatenation function $c(.)$ before entering the neural network. It is mathematically represented as:

$$x = c(f_1, \dots, f_n) = \left[v(f_{1,}), \dots v(f_n)\right]$$

where $f_1, \dots, f_n$ represent the core features and $v(f_{1,}), \dots v(f_n)$ their respective embedding function. The system performs this concatenation internally so that, depending on the types of features that are parameterised in the input, the concatenation is transparently brought about by the system. The relevance of each feature is treated in the features section.

3.6 Examples

Looking at the big picture, one could say that the ultimate goal of the algorithm is to choose the most appropriate preposition among n numbers of prepositions given a learner context, e.g. learner text such as phrases, clauses, sentences, etc. Thus, each prepositional instance within the NUCLE corpus is a potential example for testing and training the algorithm. The easiest way to create an example is to find a preposition – given its POS tag – and make the surrounding words its context. However, many factors have to be thought through: how many words around the target preposition are adequate? Should the examples go beyond the limits of the sentence, i.e. including words from neighbour sentences? Should the target preposition be included within the example?

As discussed in the chapter on prepositions, a preposition is normally followed by a noun or noun phrase that is its complement, so that the dependency between the preposition and its complement may be short, e.g. the preposition *"of"* and its complement *"humanity"* in the example *"the long-standing history <u>of</u> humanity",* or the preposition *"in"* and its complement *"changing the civilization"* in the example *"... a crucial role <u>in</u> changing the civilization..."* It would make sense then to choose a short window around the word as the example context, but the long-distance dependency problem cannot be forgotten. Therefore, the examples are generated with different lengths so that the dependency of prepositions on learner texts can be tested. This is in particular ideal for recurrent neural network which, in theory, can deal with arbitrary lengths of sequences. The parameter *"--seq_limit*" is used to control the limit of the window-size on each side of the target preposition. In previous approaches, the target preposition is located in the middle of the context, and the length of the window at each side is equivalent to each other. In this study, the window sizes on each side of the target preposition are allowed to be different from each other because RNN can deal with flexible sizes of sequences. The length on each side is different if the length at any side is shorter than the length of the remaining sentence, i.e. no information from neighbour sentences is added into any example even when previous or posterior sentences may have an influence on the target preposition. In the context of learner writing, it makes sense because complex structures are not commonly used according to the avoidance behaviour theory (Leacock et al. 2014).

The examples for the experiments were generated by iterating through the sentences of the NUCLE corpus in search of prepositions. Once a preposition was identified, words surrounding the preposition were taken on each side – up to the maximum length- and an example created. However, it was necessary to establish a method to determine the location of the target preposition among examples of variable length. Three different scenarios were set: including the original preposition in the example with the subsequent danger of biasing the neural network; not including the original preposition – which would create an ambiguous scenario for the simple LSTM network because the prediction is based on the full input sequence; and including a general tag: "*<tag_target_prep>*" in place of the original preposition. The parameter *"--original"* – used for this purpose - can take three values (*"include", "exclude"* and *"replace"*) whereby the system controls the type of examples the system should deploy.

In order to test the influence of the original preposition and the maximum length of the input sequence, two experiments were carried out. The length of the input sequence was progressively increased at values of 5, 10, 15 and 20 – sentences with a length larger than 40 are not the rule when working with learner writings. As for the original preposition, the system was deployed for each of the three distinct scenarios discussed above. When generating the examples, all punctuation marks were treated as single tokens. Finally, the evaluation metrics were used to assess the influence of each type of example in the training phase of the algorithm.

3.7 Features

It can be expected that the most relevant feature for the task of error correction is a lexical feature, and lexical features are always present in the neural network input. The main reason is that prepositions as well as other lexical classes depend on the meaning of their context to make sense, e.g. the phrases *"on the table"* and *"in the city"* share the same structural information if they are parsed to either a constituent tree or a dependency tree; they also share the same POS tags. Thus, the use of the prepositions *"on"* and *"in"* depends on the meaning of the noun phrases that follow them. It might be thought that using an infinite training set, lexical features would be enough to solve the prepositional correction problem. However, the practical limitation of the training examples force the use of additional features to provide the input features with sufficient information to deal with the prediction of prepositions. The use of more features is justified by the sparsity that some lexical tokens, and especially open classes such as nouns, can generate. For example, table 4 shows that the number of instances per lexical type is low for a large number of lexical types in the NUCLE corpus. As a consequence, probably, the algorithm would not be able to capture the influence of these low rate lexical features in the task of predicting a preposition. One could think that the embedding vector could alleviate this weakness, but it has to be remembered that the embedding table is generated by training data from the corpus so that these low rate lexical features probably cannot be located properly in the embedding vectorial space.

In order to test this hypothesis, the system includes a parameter to set a lower boundary for the number of instances a lexical type appears in the corpus, namely *"—lower_limit".* The default value of this parameter is zero, but if the value increases, the size of the vocabulary decreases as the lexical types that do not cross the lower boundary are replaced by a generic tag: "<unknown_tag>". To test the effect of the lower boundary, the algorithm is deployed using the default values and setting the lower limit at values of 1, 2, 4, 8 and 16 per experiment. The experiments are compared thrugh the analysis of the different metrics.

3.7.1 POS Tags

The second feature one would think of is the Part of Speech tag. POS tags provide general information about the example under analysis and it could be an important complement to alleviate the sparsity of some lexical features. This feature is parameterized by the argument *"--tags",* which is a Boolean that defines if the POS feature is used or not. Additionally, if this feature is set at True, the size of the embedding should be defined through the argument *"--tag_emb_size"* (unless the type of input is *"one_hot"*). To test the influence of this feature, the algorithm was deployed using the default values and the tag parameters switched to True. Then, the embedding size of the feature was tested at values of 10, 15, 20 and 25 per experiment. How the POS feature was obtained was described in The Data section.

3.7.2 Index of the Parent in the Dependency Tree

Moreover, the index of the parent in the dependency tree was also tested. Although this is a numerical value, it was treated as all other features, namely as a dense vector. The parameter that defines the presence of this feature in the model is *"--parent_index"* (True/False) and the one that defines the embedding size is *"—index_emb_size".* This feature was tested similarly to the POS tag feature. Although in the literature many other features such as dependency parse, constituency parse, lemma, semantic, etc. are used, they are outside the scope of this study, in part because recurrent neural networks keeps the structure of the sentence internally and in part because of the time constrains of this project.

3.8 Data Pre-processing

Although, these days, various neural networks models argue that no data pre-processing tasks are needed before introducing the data into the network (Liu & Liu 2017), it is usual to run several pre-processing tasks over the data before accessing the core algorithm. Most of the systems in the shared tasks performed some pre-processing before introducing the data into the systems. One of the most popular tasks among these pre-processing tasks is to correct spelling, which is a healthy measure as spelling errors and other grammatical errors generate noise that affects the performance of the error correction tasks (Dahlmeier, Ng & Ng 2012; Rozovskaya, Sammons & Roth 2012). Some methods to correct spelling errors involve the use of external tools such as Jazzy (Rozovskaya, Sammons & Roth 2012) which, given an English word, returns a list of possible words sorted by likelihood. The drawback of this approach is that there is no context analysis to help choose the right word, so this ends up by introducing erroneous words for the given context. In this study spelling correction was carried out using the Python library "autocorrect". In the context of learner data, this pre-processing task makes sense as learners are prone to misspell words and the lexical representation of some words could become sparse. Nonetheless, the impact of correcting the spelling of words without performing context analysis has to be tested carefully because a wrong correction of words could end up introducing wrong features. Thus, an experiment to compare the effectiveness of correcting spelling of isolated words was carried out by comparing the different metrics of the algorithm using the default algorithm while switching the parameter "--spelling" (which controls the use of the library to correct the spelling of the input words) on and off.

4 Results

All experiments were carried out varying only one specific element or parameter of the algorithm while all other parameters kept a constant value. The default values are shown in table 5.

Parameter	**Default**
Hidden Cells	400
Number of layers	1
Model	simple
Input Type	embedding
Word Embedding Size	100
Embedding Window	2
Number of Classes	10
Spelling	False
POS Tags	False
Tags Embedding Size	20
Index of Parent in Dependency Tree	False
Index Embedding Size	10
Lower Limit for Vocabulary	0
Oversampling	False
Oversampling Position	0
Undersampling	False
Undersampling Position	0
Length of Sequence Input (at each side)	5
Original Preposition	exclude

Table 5 Default Parameters of the algorithm

The validation dataset was used to cross validate the training phase in order to define the optimum number of epochs whereby the algorithm learns to best distinguish the different classes without losing its capacity to generalize. In practice, it is given by the point where the validation dataset gets the lowest loss value. It is helpful to verify that the algorithm is neither underfitting nor overfitting, and for each experiment, the loss rate is drawn against the number of steps (epochs), together with the accuracy of the classifier for both datasets: training and validation. The training accuracy refers to the number of examples correctly classified over the total number of examples, and must not be mistaken for the accuracy of the correction task, i.e. the training accuracy refers to the capacity of the classifier to distinguish among the classes rather than the capacity to detect and correct prepositional errors.

The testing dataset is used to verify the capability of the classifier to both distinguish among the different classes and to detect and correct prepositional errors. Two types of charts are presented to validate the algorithm's capability to separate the different classes, namely a precision-recall chart and a Receiver operating characteristic (ROC) chart. Because the algorithm is a multiclass classifier, each graph is generated using a one-vs.-all approach per class. As for the detection and correction task, for each parameter under experimentation a comparative table is presented in which all metrics (without using any threshold) are presented. The results for the task of detection are presented separately from the result for the task of correction. Moreover, because the number of prepositions is large and not all are tackled by the classifier but included in the testing dataset, an additional couple of tables in which the algorithm only addresses the classes it was designed for are presented in order to verify the behaviour of the algorithm when dealing with the right classes. The correction task for all prepositions is complemented by a chart in which all metrics are drawn against a threshold for each experiment.

Once all parameters are tested and the "optimum" algorithm is defined, a new dataset is used, namely a production dataset that was not used to test any experiment and whose data was not used to train the embedding layer hence it is the real test of the algorithm.

4.1 Size of the Neural Network

4.1.1 Number of Hidden Cells

For the number of hidden cells, the algorithm was tested using the default parameters and setting the number of hidden cells at 50, 100, 200, 400, 800 and 1600 for each experiment.

4.1.1.1 Training Phase

Following are the charts representing the training phase for each parameter:

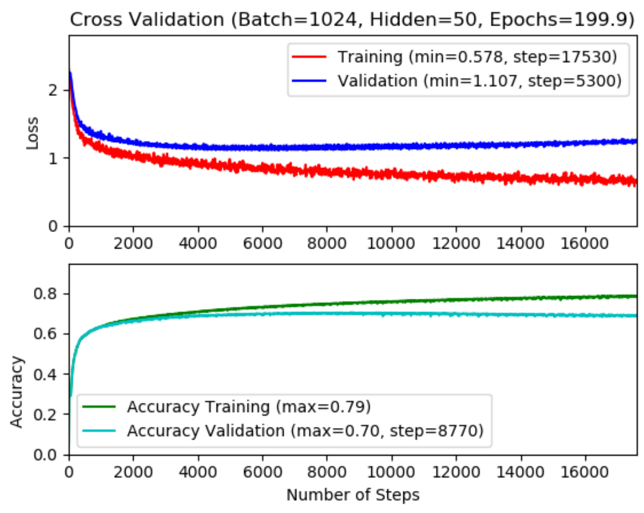

Figure 5 Cross validation using 50 Hidden Cells.

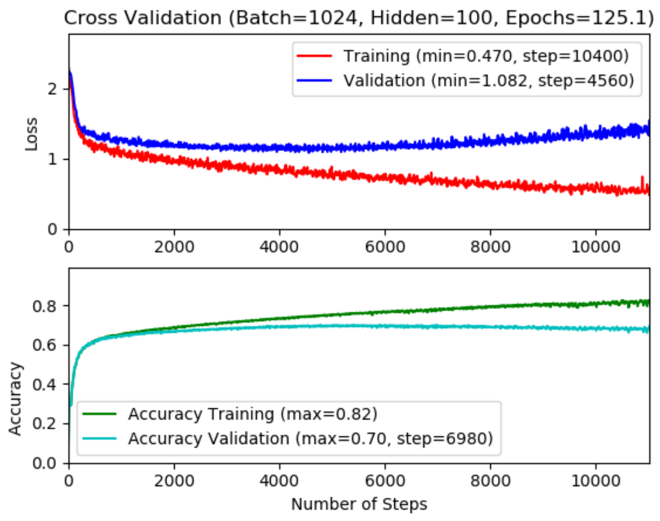

Figure 6 Cross validation using 100 Hidden Cells

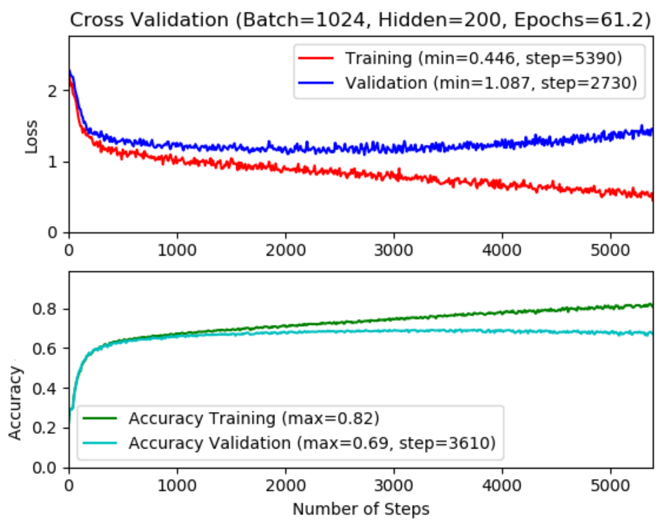

Figure 7 Cross validation using 200 Hidden Cells

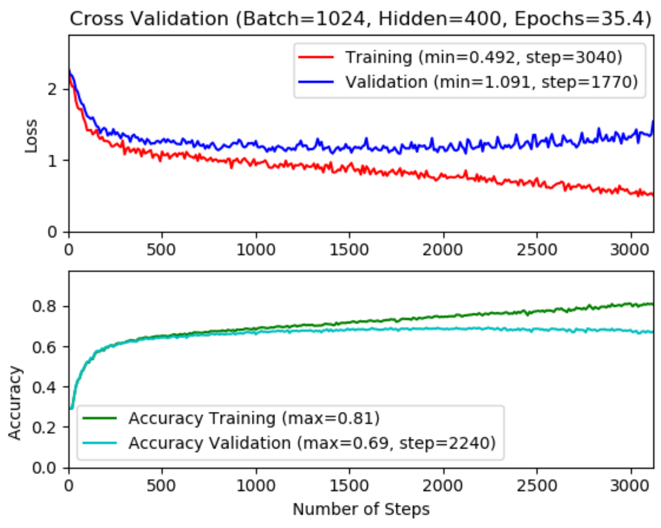

Figure 8 Cross validation using 400 Hidden Cells

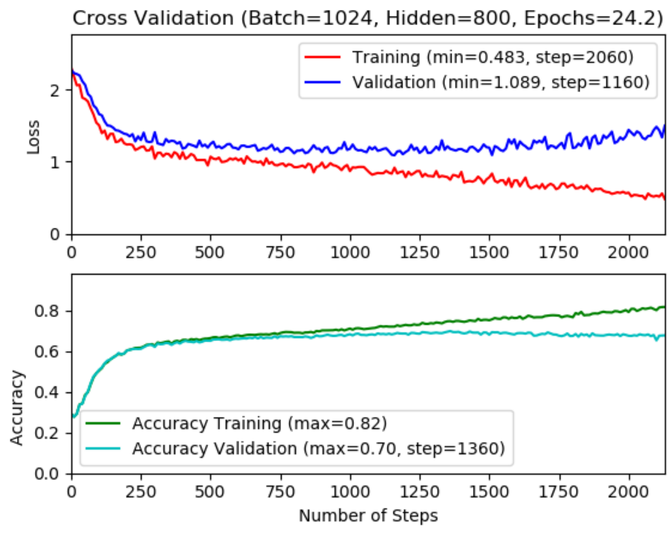

Figure 9 Cross validation using 800 Hidden Cells

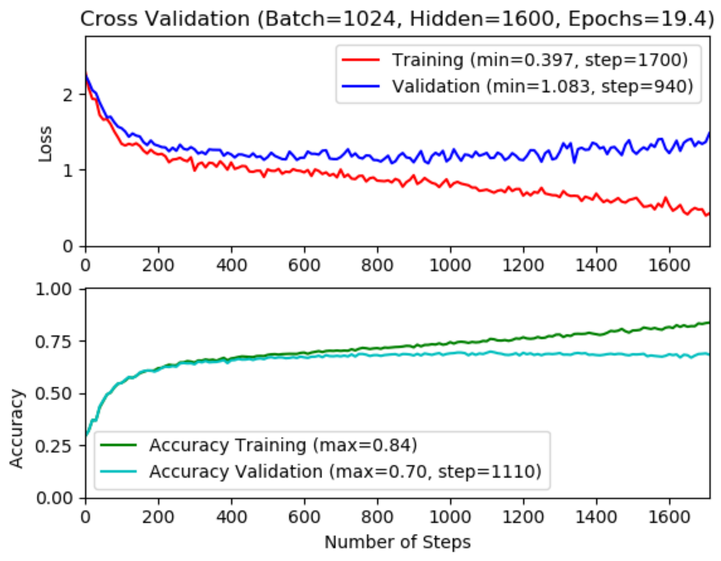

Figure 10 Cross validation using 1600 Hidden Cells

4.1.1.2 Testing Phase

4.1.1.2.1 Precision – Recall Charts

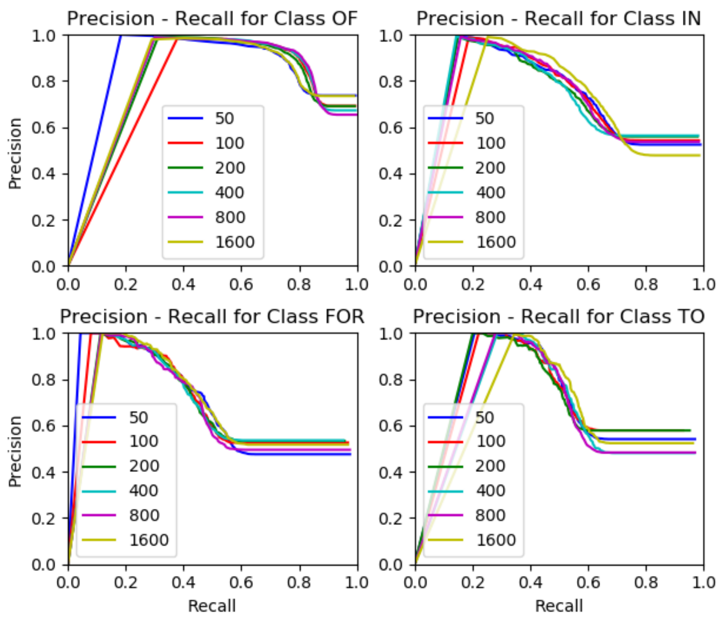

Figure 11 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the number of Hidden Cells

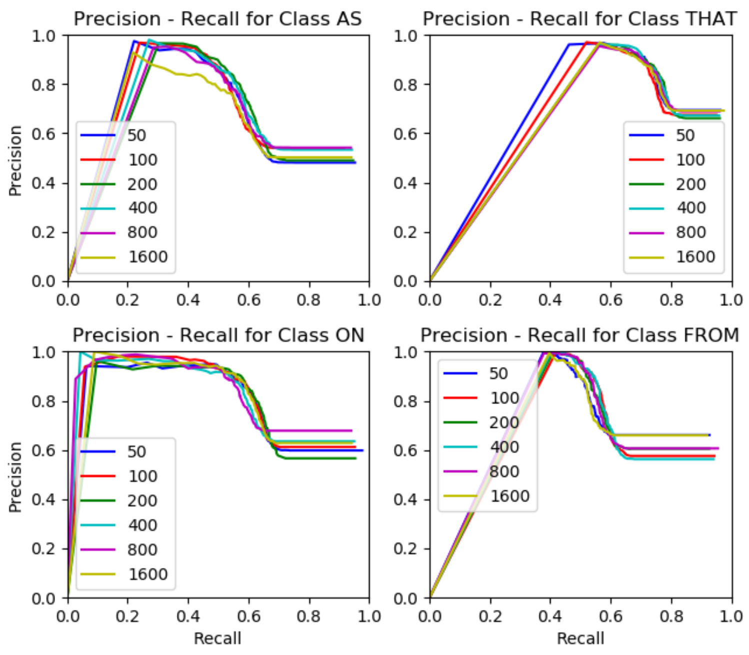

Figure 12 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the number of Hidden Cells

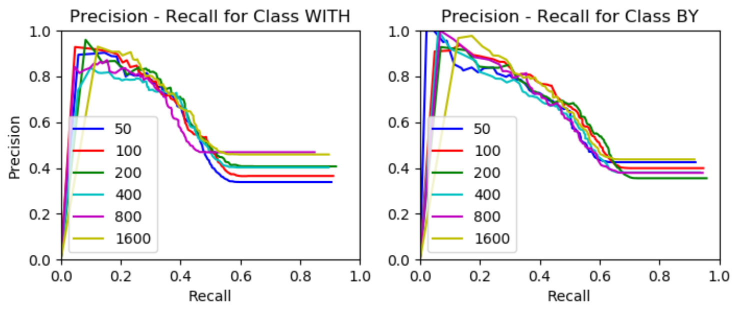

Figure 13 Precision vs. Recall for classes WITH (left), BY (right) varying the number of Hidden Cells

4.1.1.2.2 ROC Charts

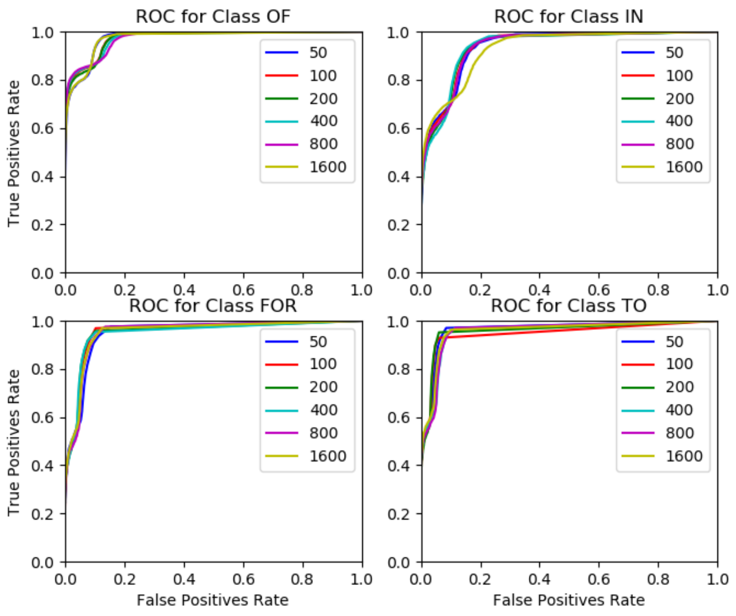

Figure 14 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the number of Hidden Cells

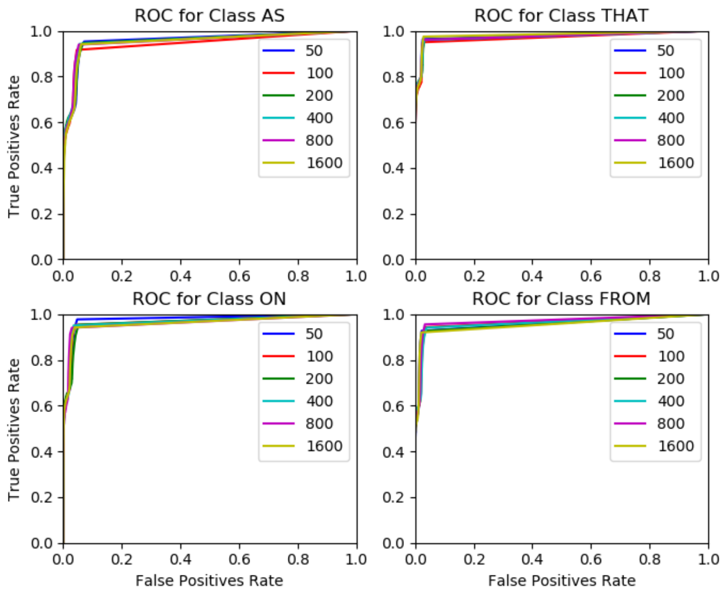

Figure 15 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the number of Hidden Cells

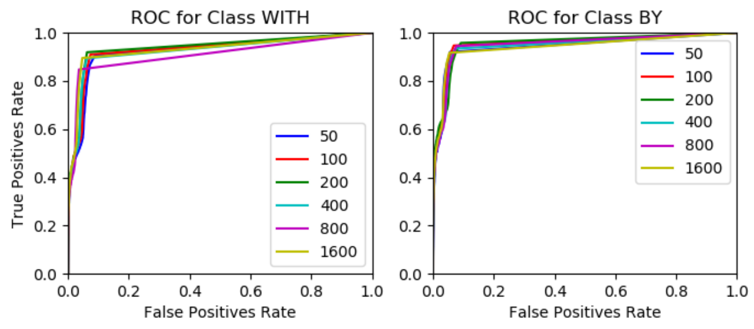

Figure 16 ROC for classes WITH (left) and BY (right) varying the number of Hidden Cells

4.1.1.2.3 Metrics for Error Correction against Threshold

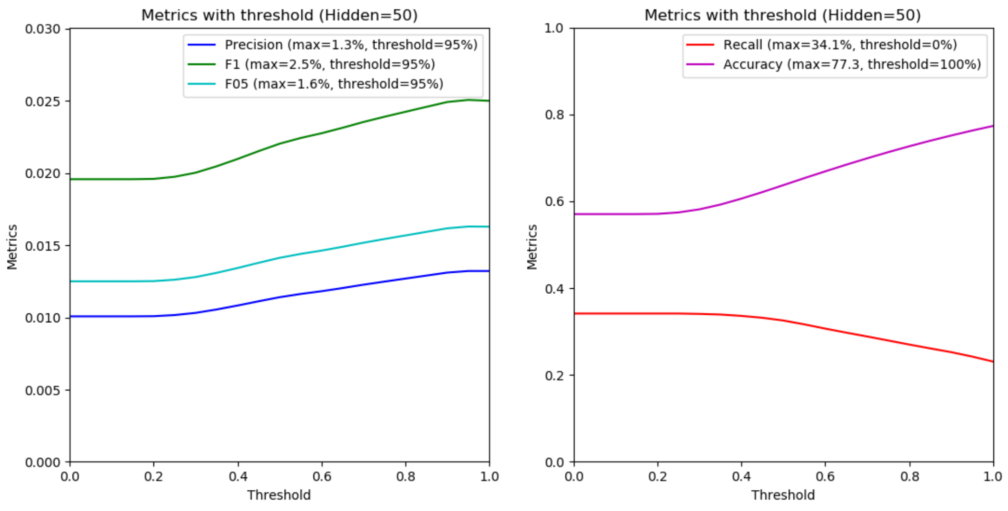

Figure 17 Correction Metrics against a Threshold using 50 Hidden Cells

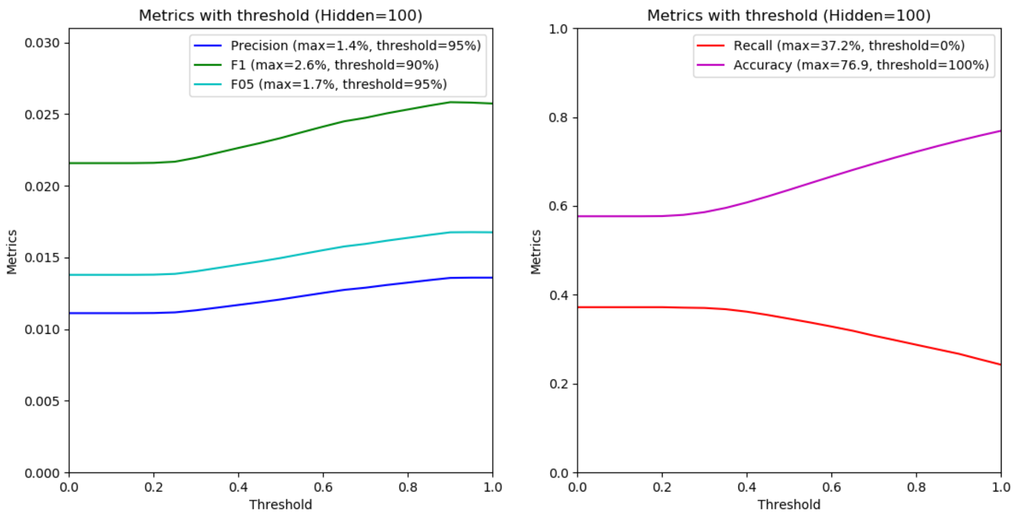

Figure 18 Correction Metrics against a Threshold using 100 Hidden Cells

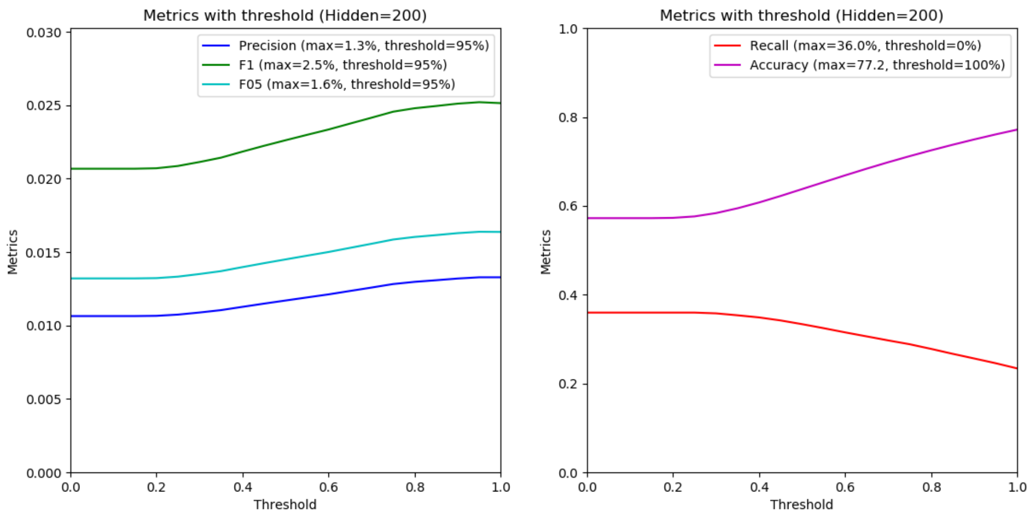

Figure 19 Correction Metrics against a Threshold using 200 Hidden Cells

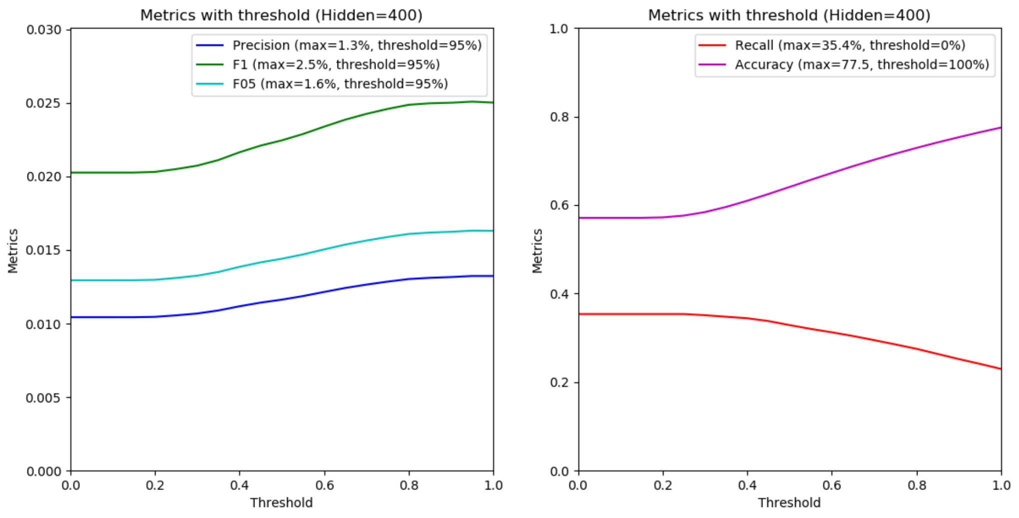

Figure 20 Correction Metrics against a Threshold using 400 Hidden Cells

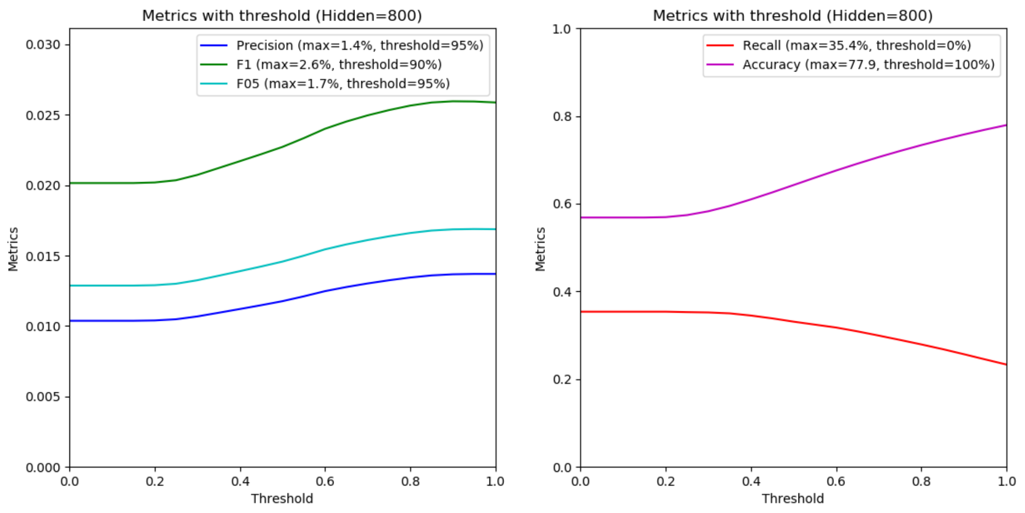

Figure 21 Correction Metrics against a Threshold using 800 Hidden Cells

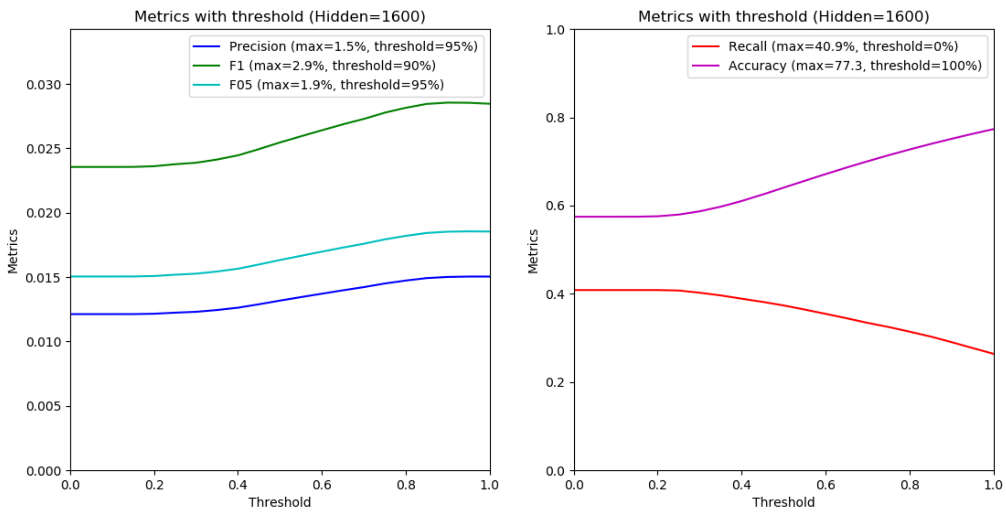

Figure 22 Correction Metrics against a Threshold using 1600 Hidden Cells

4.1.1.2.4 Metrics for Error Detection and Correction Tasks

Hidden Cells	Precision	Recall	F1	F05	Accuracy
50	1.00%	34.15%	1.96%	1.25%	57.00%
100	1.11%	37.19%	2.16%	1.38%	57.65%
200	1.06%	35.97%	2.07%	1.32%	57.23%
400	1.10%	37.8%	2.14%	1.37%	56.64%
800	1.03%	35.36%	2.01%	1.29%	56.83%
1600	1.21%	40.85%	2.35%	1.50%	57.47%

Table 6 Error correction for all classes using different numbers of hidden cells

Hidden Cells	Precision	Recall	F1	F05	Accuracy
50	2.12%	75.0%	4.30%	2.74%	57.85%
100	2.37%	79.27%	4.60%	2.94%	58.49%
200	2.34%	79.27%	4.56%	2.91%	58.09%
400	2.4%	82.32%	4.66%	2.98%	57.53%
800	2.34%	79.88%	4.55%	2.91%	57.71%
1600	2.37%	79.87%	4.61%	2.94%	58.24%

Table 7 Error detection for all classes using different numbers of hidden cells

Hidden Cells	Precision	Recall	F1	F05	Accuracy
50	1.60%	40.87%	3.06%	1.97%	67.74%
100	1.73%	44.52%	3.39%	2.18%	68.46%
200	1.68%	43.06%	3.24%	2.08%	67.96%
400	1.73%	45.25%	3.33%	2.14%	67.27%
800	1.63%	42.33%	3.14%	2.02%	67.49%
1600	1.92%	48.90%	3.70%	2.37%	68.26%

Table 8 Error correction for 10 top classes using different numbers of hidden cells

Hidden Cells	Precision	Recall	F1	F05	Accuracy
50	3.6%	78.83%	5.90%	3.79%	68.54%
100	3.36%	84.67%	6.46%	4.15%	69.30%
200	3.22%	82.48%	6.21%	3.99%	68.80%
400	3.34%	87.59%	6.44%	4.13%	68.15%
800	3.23%	83.94%	6.23%	4.00%	68.36%
1600	3.30%	83.94%	6.34%	4.08%	68.70%

Table 9 Error detection for top 10 classes using different numbers of hidden cells

4.1.2 Number of Layers

For the number of layers (depth of the neural network), the algorithm was tested using the default parameters and setting the number of layers at 2, 4, 8. The experiment using 1 layer was already carried out in the number of hidden cells section.

4.1.2.1 Training Phase

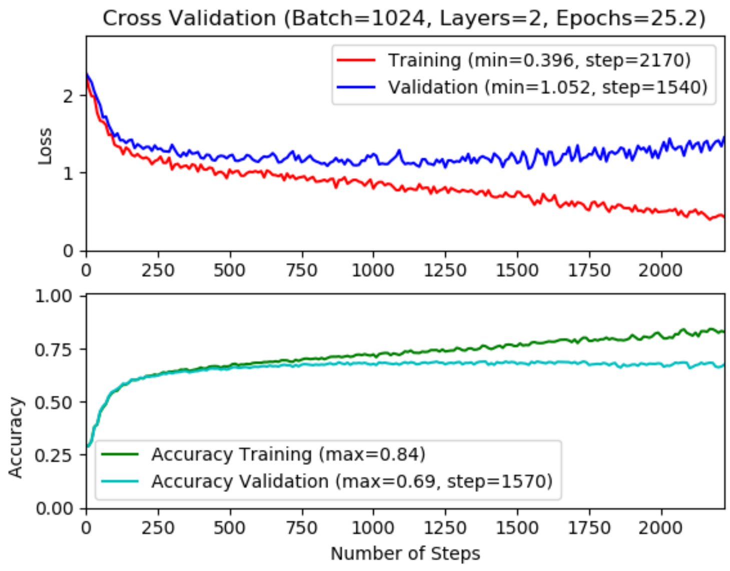

Figure 23 Cross validation using 2 Layers

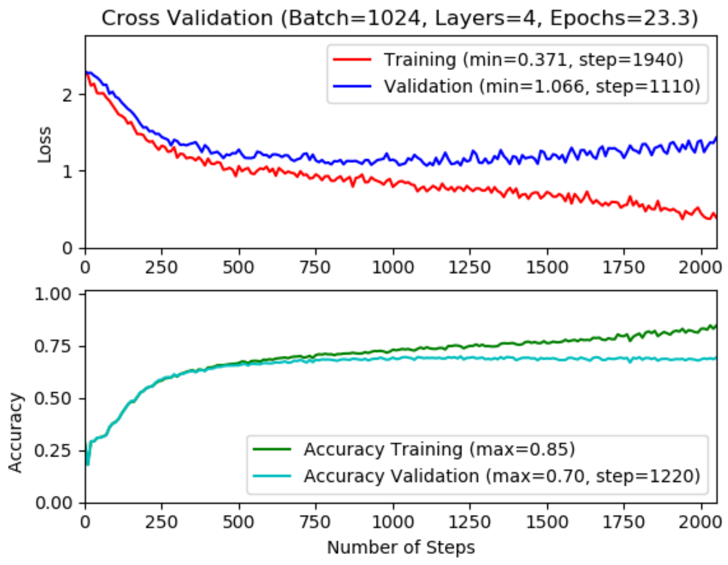

Figure 24 Cross validation using 4 Layers

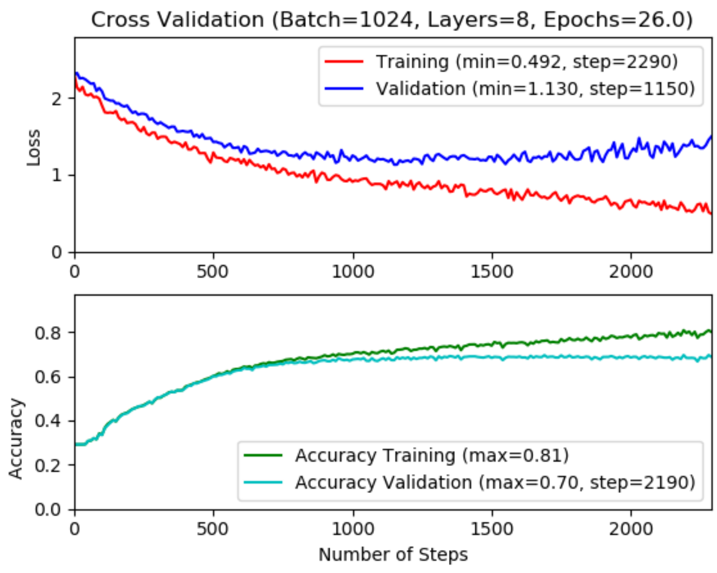

Figure 25 Cross validation using 8 Layers

4.1.2.2 Testing Phase

4.1.2.2.1 Precision – Recall Graphs

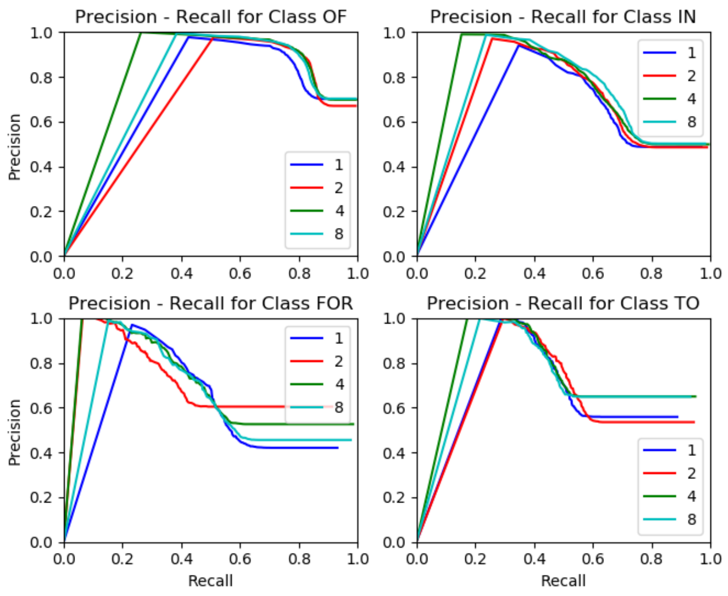

Figure 26 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the number of layers

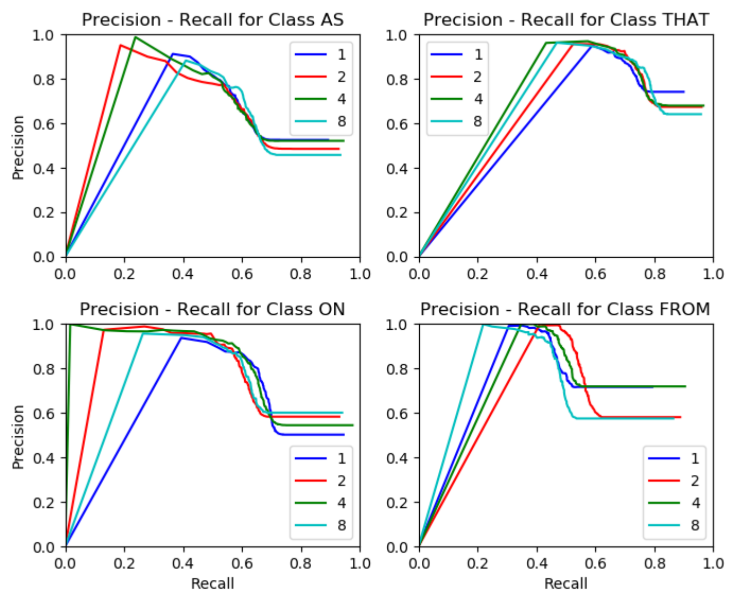

Figure 27 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the number of layers

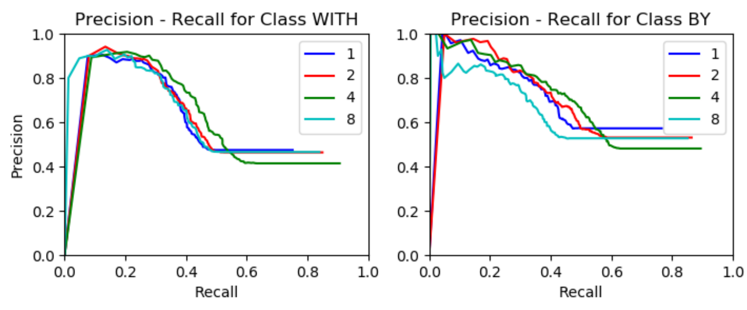

Figure 28 Precision vs. Recall for classes WITH (left) and BY (right) varying the number of layers

4.1.2.2.2 ROC Graphs

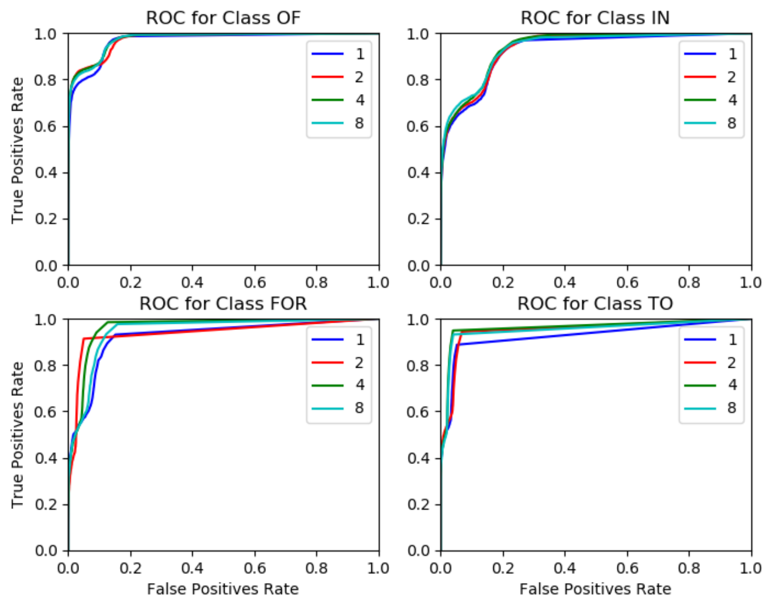

Figure 29 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the number of Layers

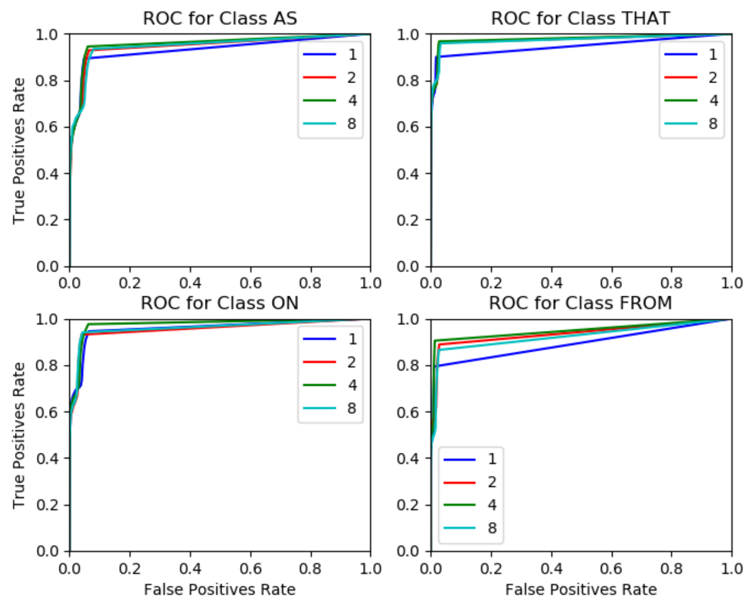

Figure 30 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the number of layers

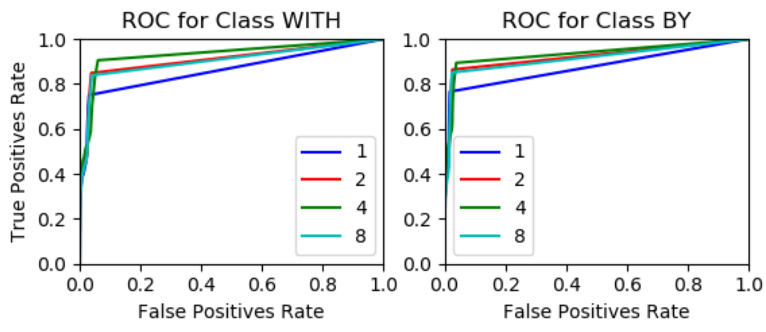

Figure 31 ROC for classes WITH (left) and BY (right) varying the number of layers

4.1.2.2.3 Metrics for Error Correction against a Threshold

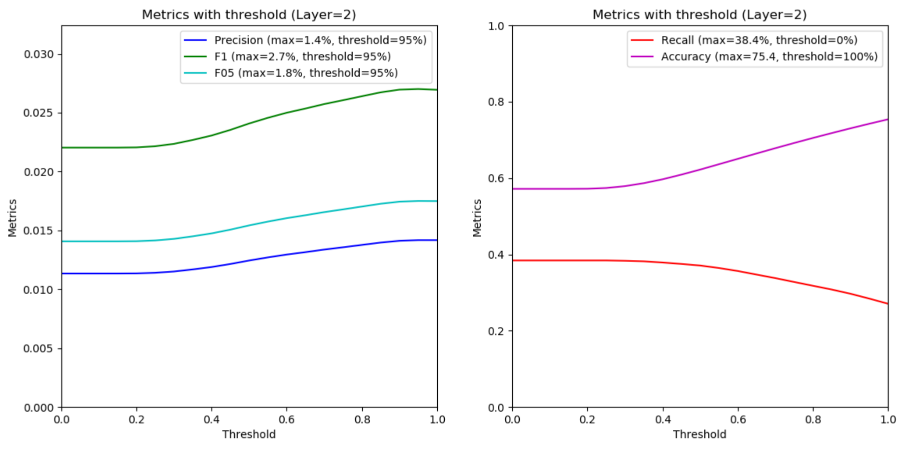

Figure 32 Correction Metrics against a Threshold using 2 Layers

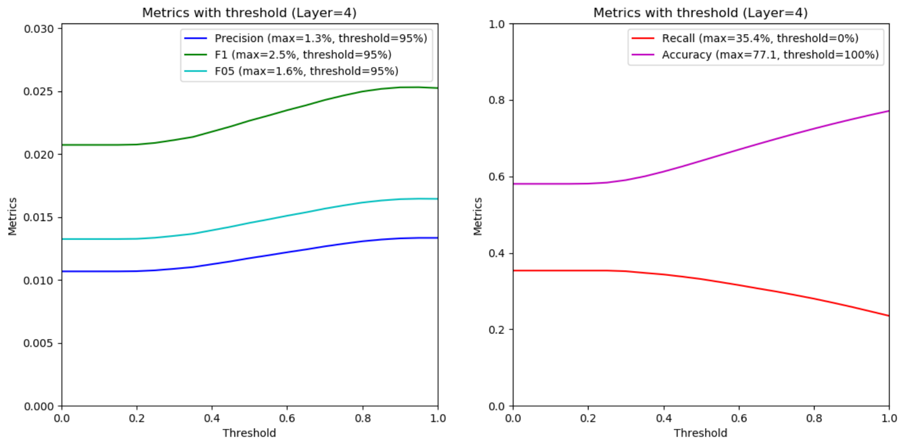

Figure 33 Correction Metrics against a Threshold using 4 Layers

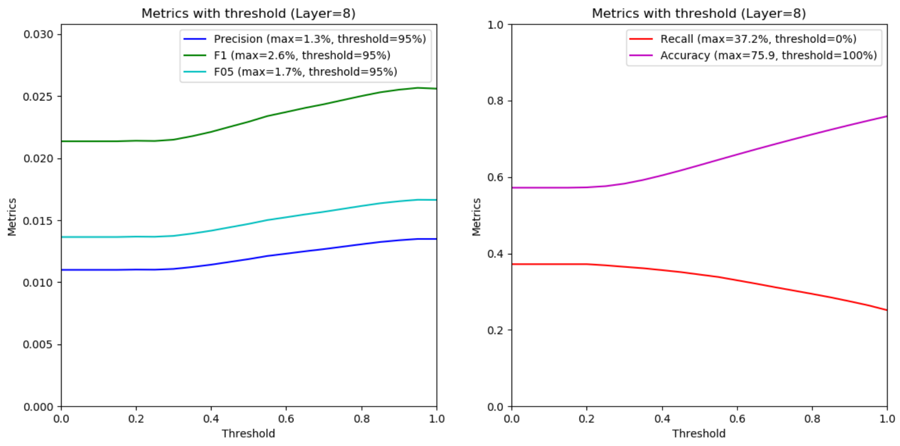

Figure 34 Correction Metrics against a Threshold using 8 Layers

4.1.2.2.4 Metrics for Error Detection and Correction Tasks

Number of Layers	Precision	Recall	F1	F05	Accuracy
1	1.10%	37.8%	2.14%	1.37%	56.64%
2	1.13%	38.4%	2.20%	1.40%	57.17%
4	1.07%	35.4%	2.07%	1.32%	58.05%
8	1.09%	37.20%	2.14%	1.36%	57.20%

Table 10 Error correction for all classes using different numbers of layers

Number of Layers	Precision	Recall	F1	F05	Accuracy
1	2.4%	82.32%	4.66%	2.98%	57.53%
2	2.30%	78.05%	4.47%	2.86%	57.96%
4	2.32%	76.83%	4.50%	2.88%	58.88%
8	2.27%	76.82%	4.41%	2.82%	57.99%

Table 11 Error detection for all classes using different numbers of layers

Number of Layers	Precision	Recall	F1	F05	Accuracy
1	1.73%	45.25%	3.33%	2.14%	67.27%
2	1.79%	45.98%	3.44%	2.16%	67.89%
4	1.71%	42.33%	3.28%	2.11%	68.94%
8	1.74%	44.52%	3.35%	2.15%	67.95%

Table 12 Error correction for 10 top classes using different numbers of layers

Number of Layers	Precision	Recall	F1	F05	Accuracy
1	3.34%	87.59%	6.44%	4.13%	68.15%
2	3.24%	83.21%	6.24%	4.01%	68.67%
4	3.24%	80.29%	6.22%	4.01%	69.74%
8	3.10%	79.56%	5.98%	3.84%	68.68%

Table 13 Error detection for 10 top classes using different numbers of layers

4.2 Data

4.2.1 Undersampling

For undersampling the examples of the training data (validation and testing were not modified), the *"under"* parameter was set at True and the under position was set at values of 2, 3, 7 and 10.

4.2.1.1 Training Phrase

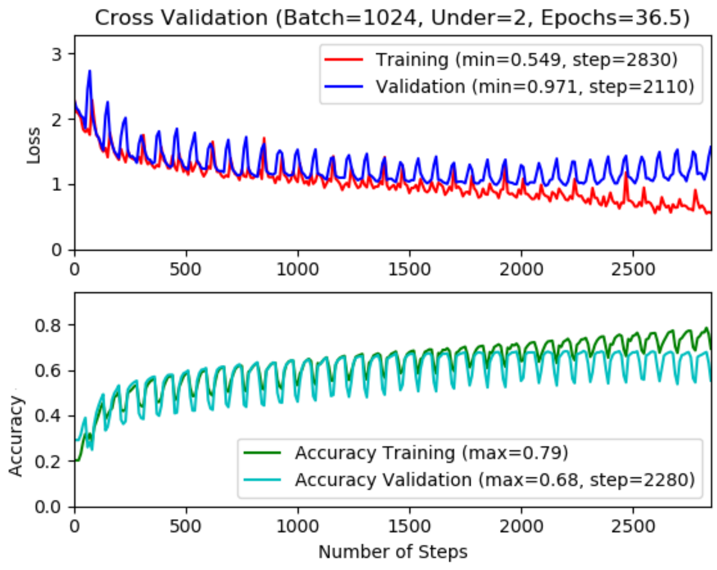

Figure 35 Cross validation by undersampling the maximum number of instances per class from the second most frequent preposition in the corpus, i.e. 'IN'

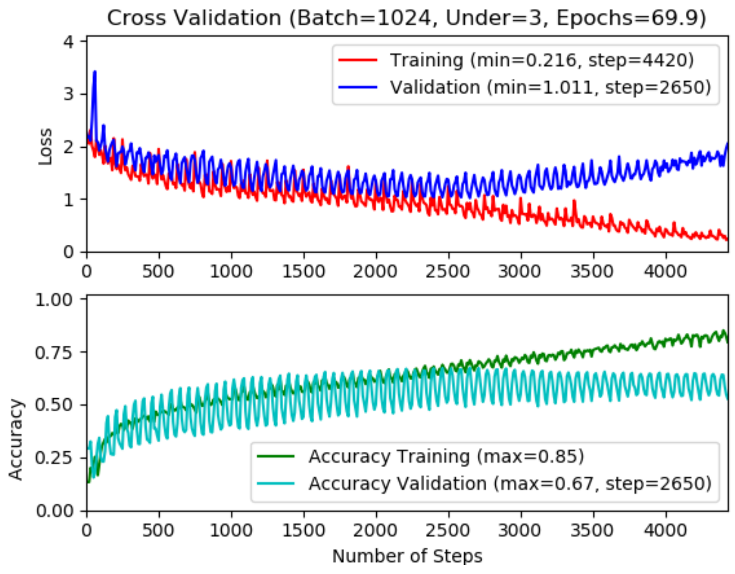

Figure 36 Cross validation by undersampling the maximum number of instances per class from the third most frequent preposition in the corpus, i.e. 'FOR'

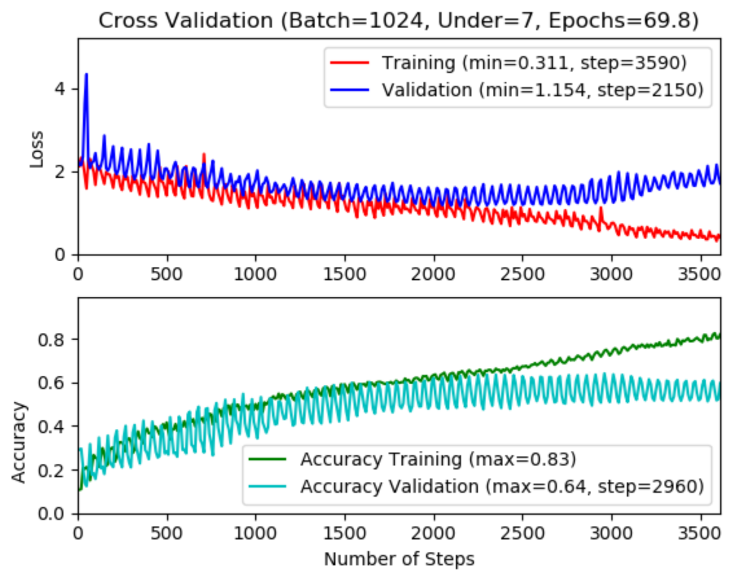

Figure 37 Cross validation by undersampling the maximum number of instances per class from the seventh most frequent preposition in the corpus, i.e. 'ON'

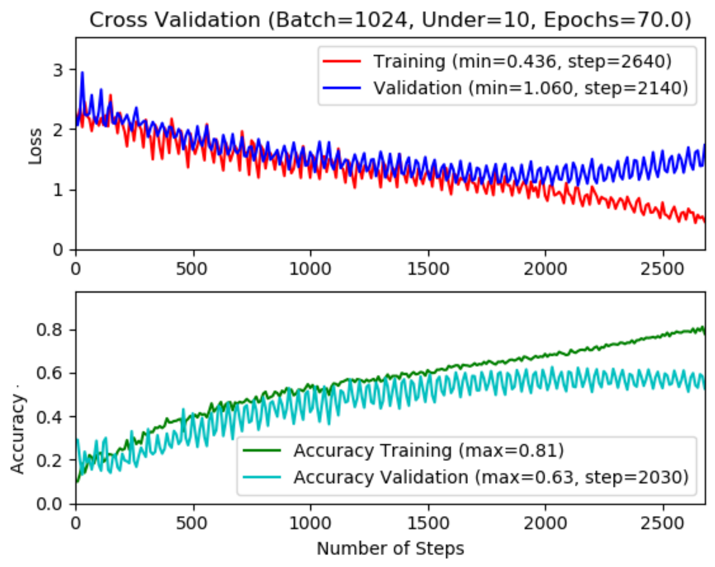

Figure 38 Cross validation by undersampling the maximum number of instances per class from the tenth most frequent preposition in the corpus, i.e. 'BY'

4.2.1.2 Testing Phase

4.2.1.2.1 Precision-Recall Graphs

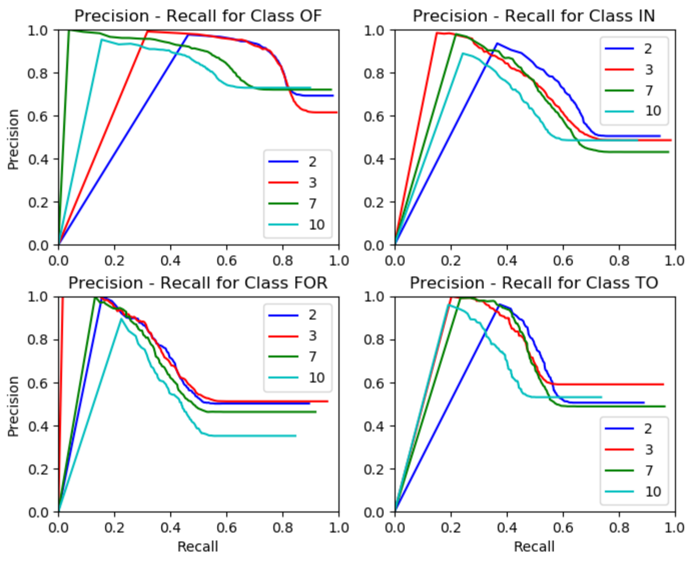

Figure 39 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) by undersampling training examples

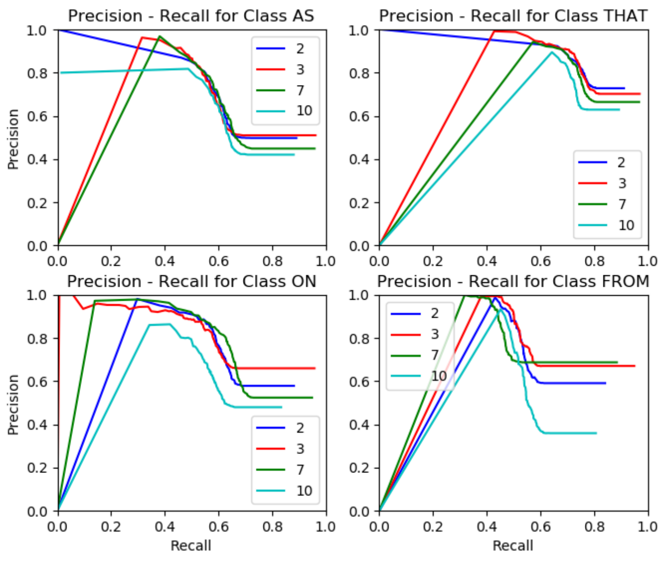

Figure 40 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) by undersampling the training examples

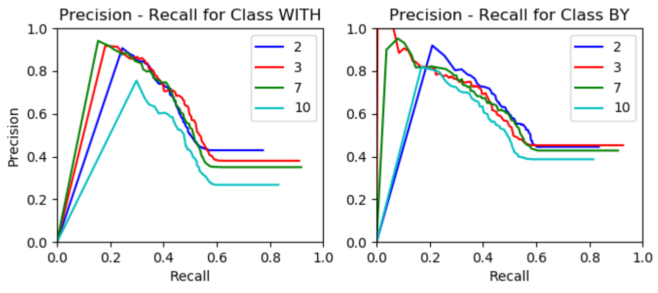

Figure 41 Precision vs. Recall for classes WITH (left) and BY (right) by undersampling the training examples

4.2.1.2.2 ROC Graphs

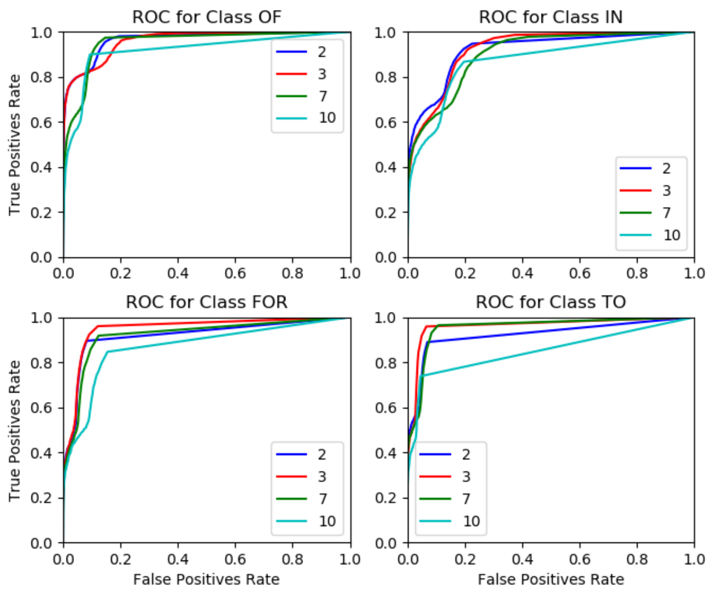

Figure 42 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) by undersampling the training examples

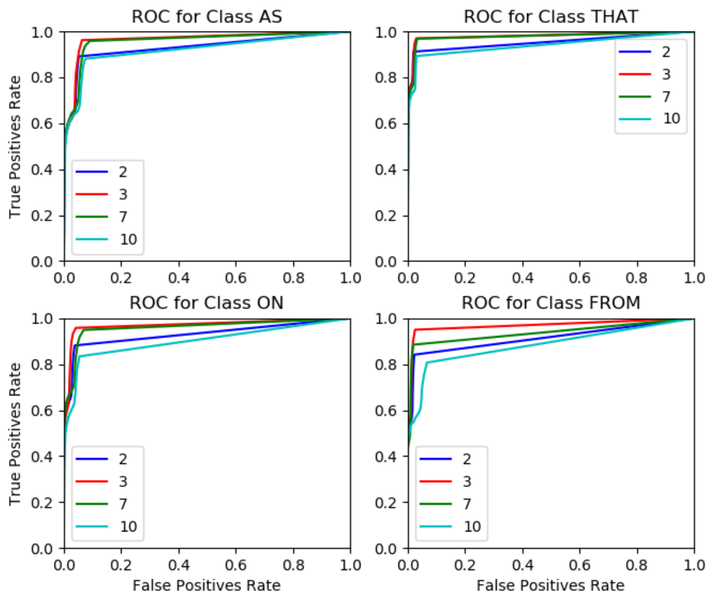

Figure 43 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) by undersampling the training examples

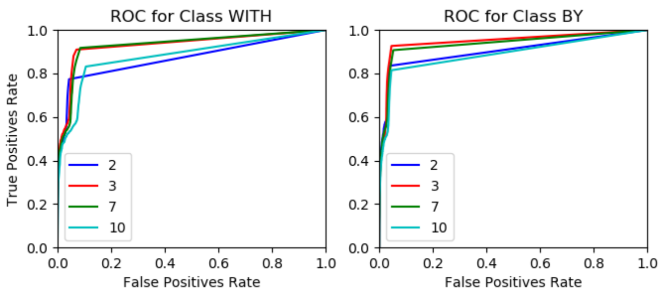

Figure 44 ROC for classes WITH (left) and BY (right) by undersampling the training examples

4.2.1.2.3 Metrics for Error Correction against a Threshold

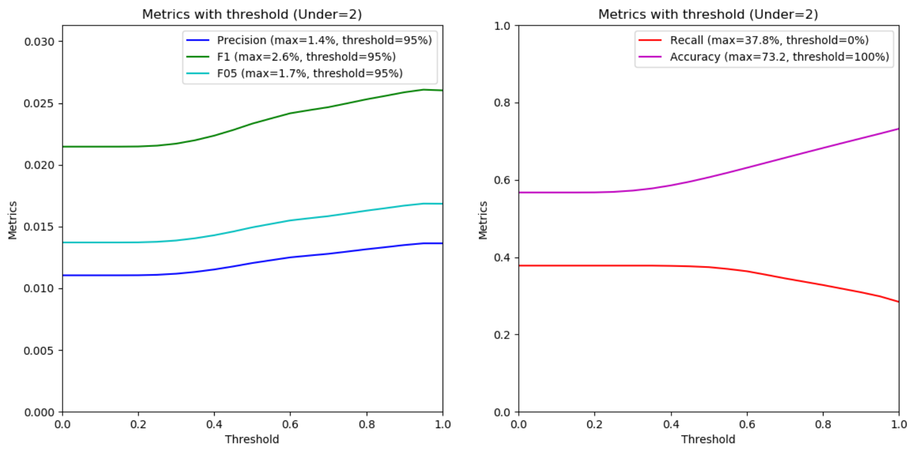

Figure 45 Correction Metrics against a Threshold by undersampling the maximum number of instances per class from the second most frequent preposition in the corpus, i.e. 'IN'

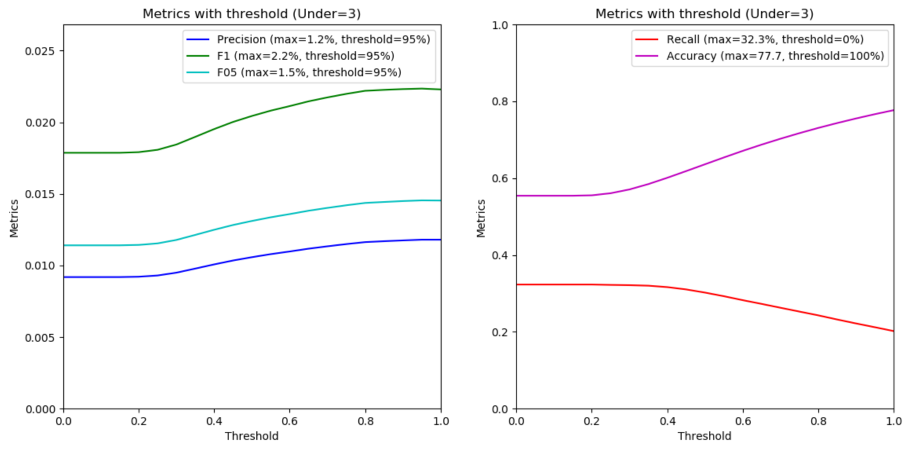

Figure 46 Correction Metrics against a Threshold by undersampling the maximum number of instances per class from the third most frequent preposition in the corpus, i.e. 'FOR'

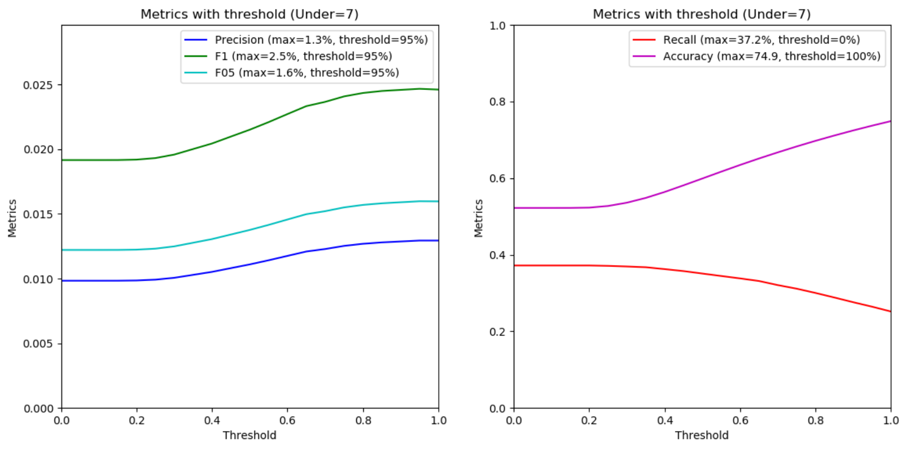

Figure 47 Correction Metrics against a Threshold by undersampling the maximum number of instances per class from the seventh most frequent preposition in the corpus, i.e. 'ON'

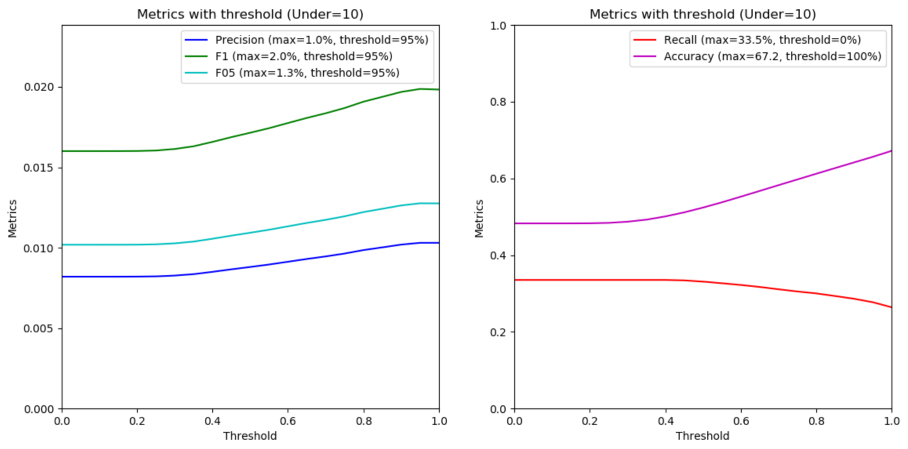

Figure 48 Correction Metrics against a Threshold by undersampling the maximum number of instances per class from the tenth most frequent preposition in the corpus, i.e. 'BY'

4.2.1.2.4 Metrics for Error Detection and Correction Tasks

Under	Precision	Recall	F1	F05	Accuracy
without	1.10%	37.8%	2.14%	1.37%	56.64%
2	1.10%	37.80%	2.15%	1.37%	56.72%
3	0.92%	32.31%	1.79%	1.14%	55.43%
7	0.98%	37.19%	1.92%	1.22%	52.22%
10	0.82%	33.54%	1.60%	1.02%	48.29%

Table 14 Error correction for all classes by undersampling the training examples

Under	Precision	Recall	F1	F05	Accuracy
without	2.4%	82.32%	4.66%	2.98%	57.53%
2	2.32%	79.27%	4.50%	2.87%	57.54%
3	2.25%	79.27%	4.38%	2.80%	56.35%
7	2.14%	81.10%	4.18%	2.66%	53.07%
10	2.08%	85.36%	4.07%	2.59%	49.26%

Table 15 Error detection for all classes by undersampling the training examples

Under	Precision	Recall	F1	F05	Accuracy
without	1.73%	45.25%	3.33%	2.14%	67.27%
2	1.73%	45.25%	3.34%	2.15%	67.38%

3	1.42%	38.69%	2.74%	1.76%	65.82%
7	1.46%	44.52%	2.84%	1.82%	62.02%
10	1.18%	40.15%	2.29%	1.46%	57.36%

Table 16 Error correction for 10 top classes by undersampling the training examples

Under	Precision	Recall	F1	F05	Accuracy
without	3.34%	87.59%	6.44%	4.13%	68.15%
2	3.11%	81.02%	5.98%	3.85%	68.14%
3	3.08%	83.94%	5.94%	3.82%	66.76%
7	2.79%	84.67%	5.39%	3.45%	62.84%
10	2.55%	86.86%	4.95%	3.16%	58.28%

Table 17 Error detection for 10 top classes by undersampling the training examples

4.2.2 Oversampling

For oversampling the examples of the training data (validation and testing were not modified), the “over” parameter was set at True and the over position was set at values of 1, 2, 3 and 7.

4.2.2.1 Training Phase

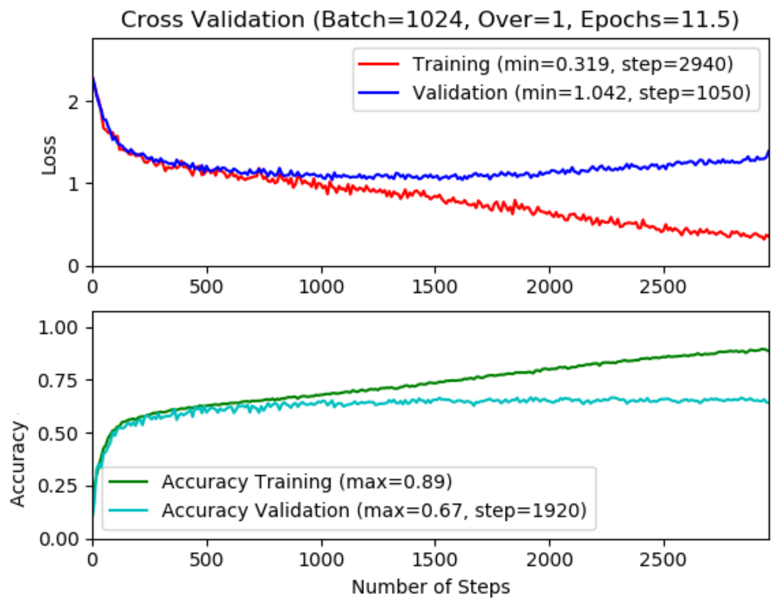

Figure 49 Cross validation by oversampling the minimum number of instances per class from the most frequent preposition in the corpus, i.e. ‘OF’

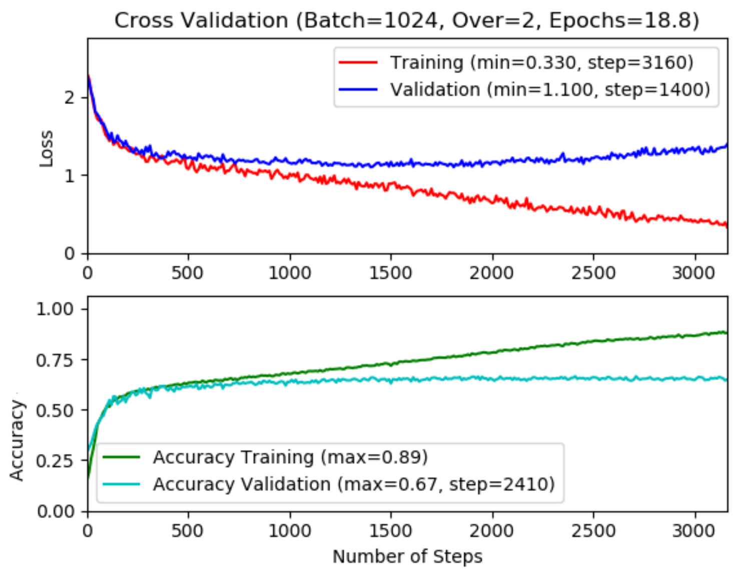

Figure 50 Cross validation by oversampling the minimum number of instances per class from the second most frequent preposition in the corpus, i.e. 'IN'

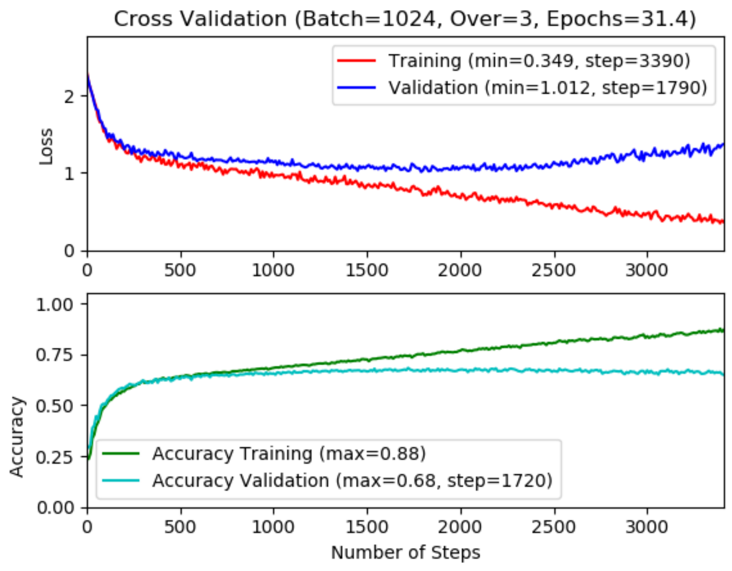

Figure 51 Cross validation by oversampling the minimum number of instances per class from the third most frequent preposition in the corpus, i.e. 'FOR'

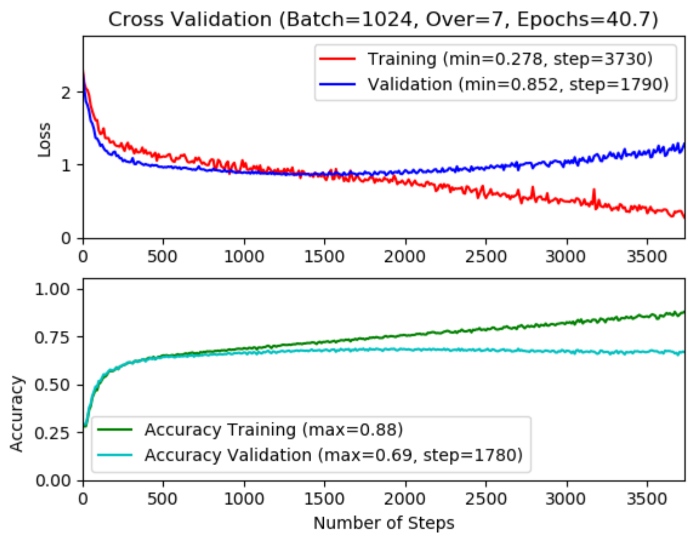

Figure 52 Cross validation by oversampling the minimum number of instances per class from the seventh most frequent preposition in the corpus, i.e. 'ON'

4.2.2.2 Testing Phase

4.2.2.2.1 Precision-Recall Graphs

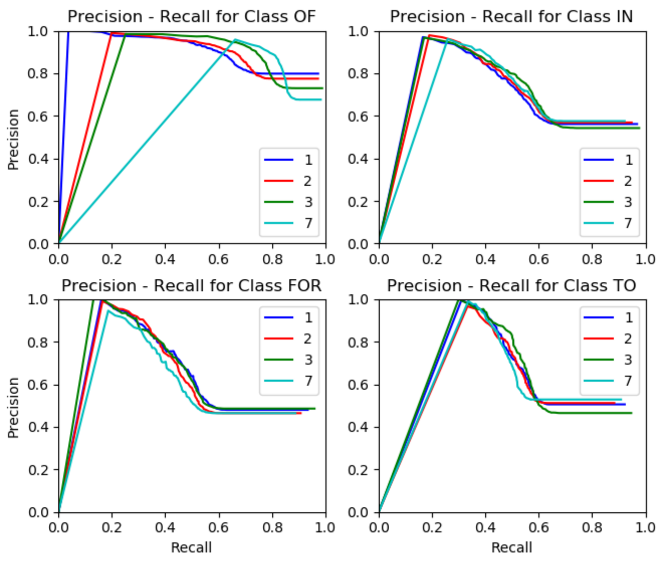

Figure 53 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) by oversampling the training examples

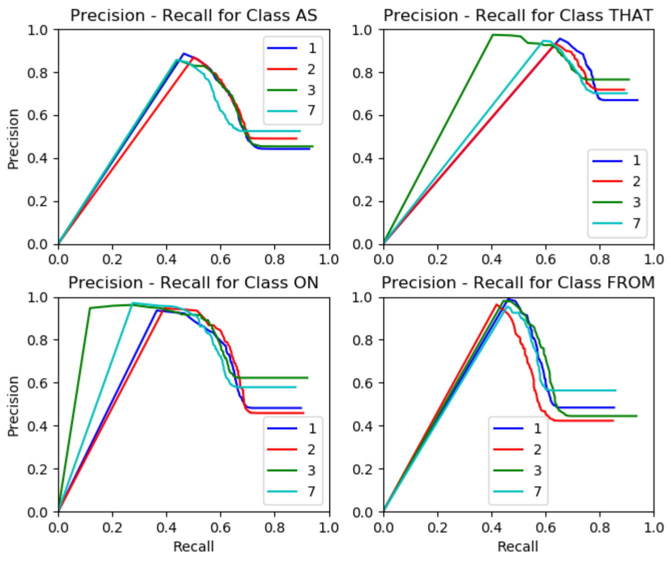

Figure 54 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) by oversampling the training examples

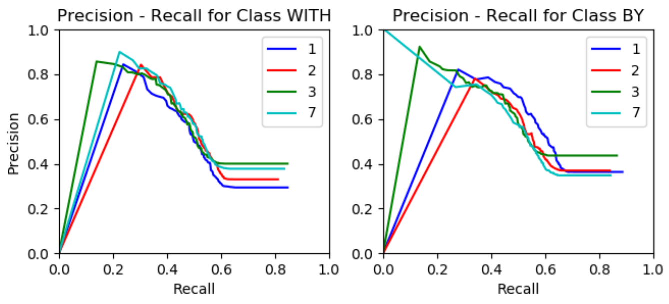

Figure 55 Precision vs. Recall for classes WITH (left) and BY (right) by oversampling the training examples

4.2.2.2.2 ROC Graphs

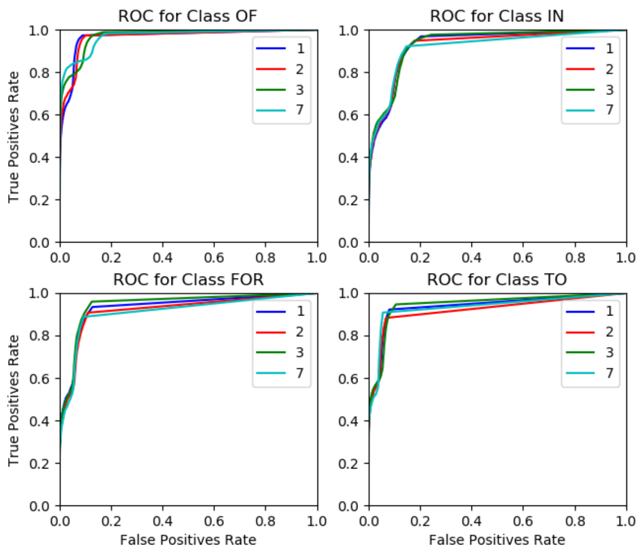

Figure 56 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) by oversampling the training examples

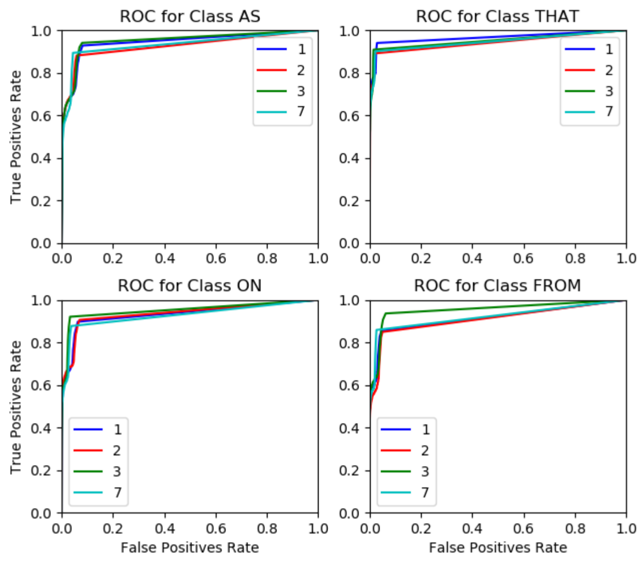

Figure 57 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) by oversampling the training examples

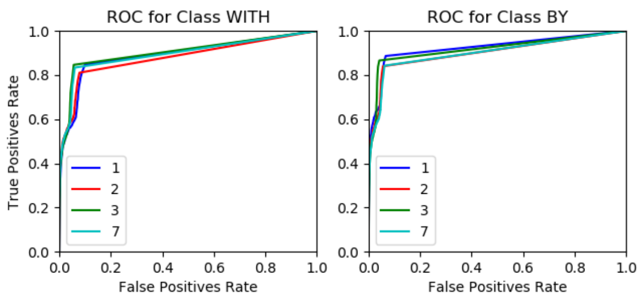

Figure 58 for classes WITH (left) and BY (right) by oversampling the training examples

4.2.2.2.3 Metrics for Error Correction against a Threshold

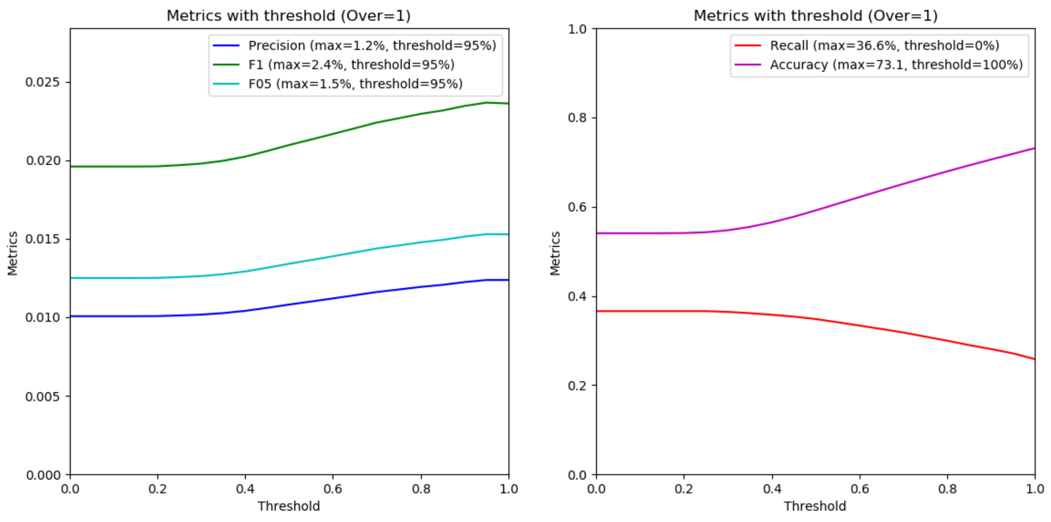

Figure 59 Correction Metrics against a Threshold by oversampling the minimum number of instances per class from the most frequent preposition in the corpus, i.e. 'OF'

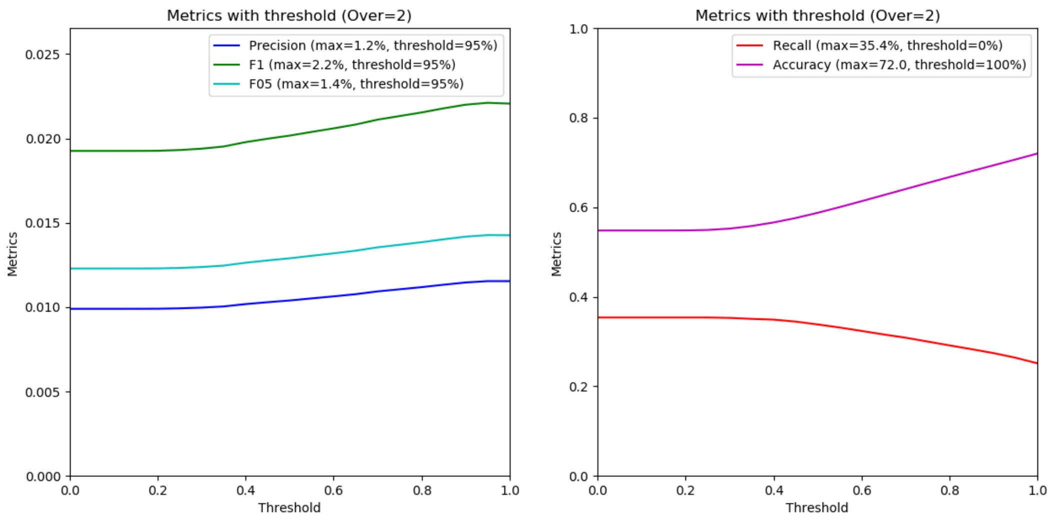

Figure 60 Correction Metrics against a Threshold by oversampling the minimum number of instances per class from the second most frequent preposition in the corpus, i.e. 'IN'

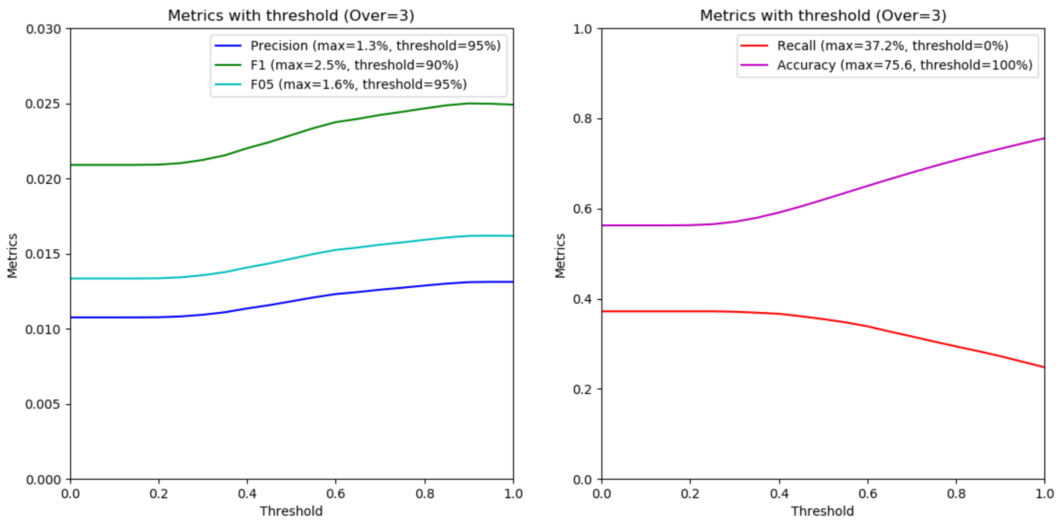

Figure 61 Correction Metrics against a Threshold by oversampling the minimum number of instances per class from the third most frequent preposition in the corpus, i.e. 'FOR'

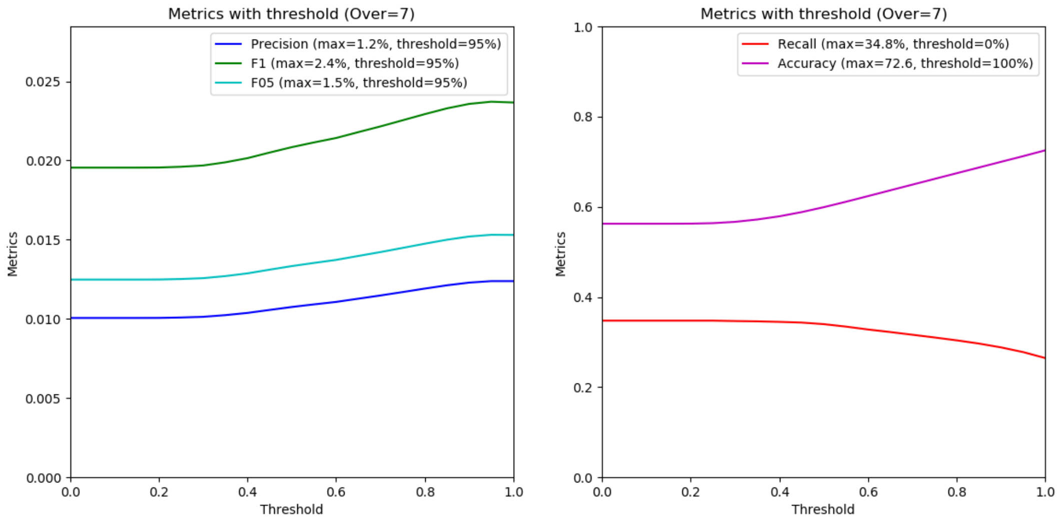

Figure 62 Correction Metrics against a Threshold by oversampling the minimum number of instances per class from the most frequent preposition in the corpus, i.e. 'ON'

4.2.2.2.4 Metrics for Error Detection and Correction Tasks

Over	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
without	1.10%	37.8%	2.14%	1.37%	56.64%
1	1.00%	36.58%	1.96%	1.25%	54.06%
2	0.99%	35.36%	1.92%	1.23%	54.80%
3	1.07%	37.19%	2.09%	1.33%	56.27%
7	1.00%	34.76%	1.95%	1.25%	56.26%

Table 18 Error correction for all classes by oversampling the training examples

Over	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
without	2.4%	82.32%	4.66%	2.98%	57.53%
1	2.31%	84.14%	4.50%	2.87%	54.98%
2	2.27%	81.10%	4.42%	2.82%	55.70%
3	2.34%	81.10%	4.56%	2.91%	57.14%
7	2.38%	82.32%	4.63%	2.95%	57.20%

Table 19 Error detection for all classes by oversampling the training examples

Over	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
without	1.73%	45.25%	3.33%	2.14%	67.27%
1	1.53%	43.78%	2.95%	1.90%	64.21%
2	1.52%	42.33%	2.93%	1.88%	65.09%
3	1.68%	44.52%	3.23%	2.08%	66.83%
7	1.57%	41.60%	3.02%	1.95%	66.83%

Table 20 Error correction for 10 top classes by oversampling the training examples

Over	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
without	3.34%	87.59%	6.44%	4.13%	68.15%
1	3.01%	86.13%	5.81%	3.73%	65.08%
2	3.01%	83.94%	5.81%	3.73%	65.95%
3	3.19%	84.67%	6.15%	3.95%	67.67%
7	3.17%	83.94%	6.11%	3.93%	67.72%

Table 21 Error correction for 10 top classes by oversampling the training examples

4.3 Examples

4.3.1 Length of the Input Sequence

For the length of the input sequence, the algorithm was tested using the default parameters and setting the sequence limit at values of 10, 15 and 20 per experiment. The experiment using a sequence limit of 5 was tested in the number of hidden cells section.

4.3.1.1 Training Phase

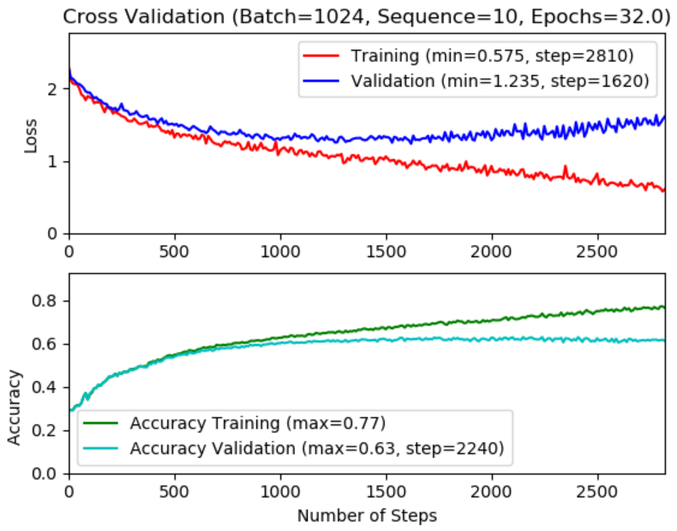

Figure 63 Table 18 Cross validation at a maximum input sequence of 10

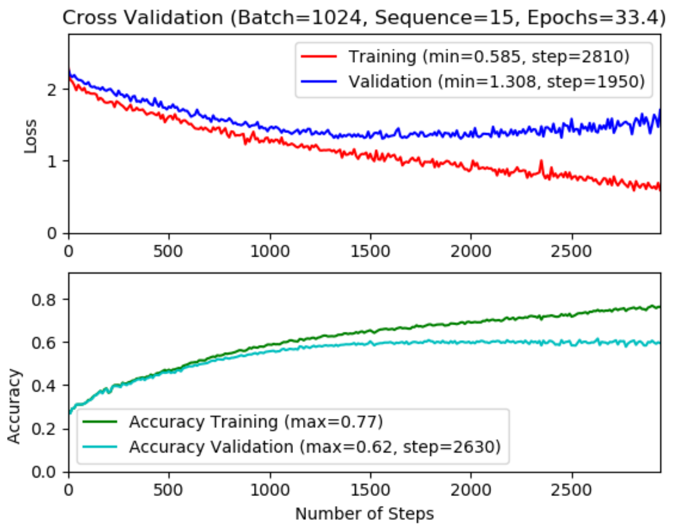

Figure 64 Table 18 Cross validation at a maximum input sequence of 15

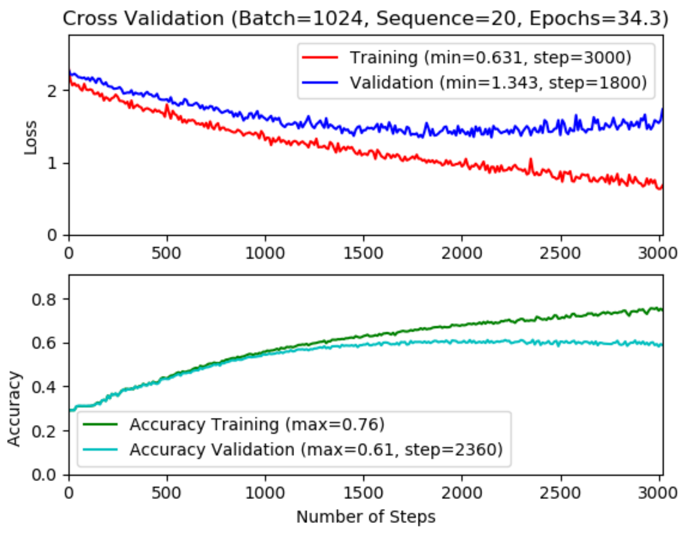

Figure 65 Table 18 Cross validation at a maximum input sequence of 20

4.3.1.2 Testing Phase

4.3.1.2.1 Precision-Recall Graphs

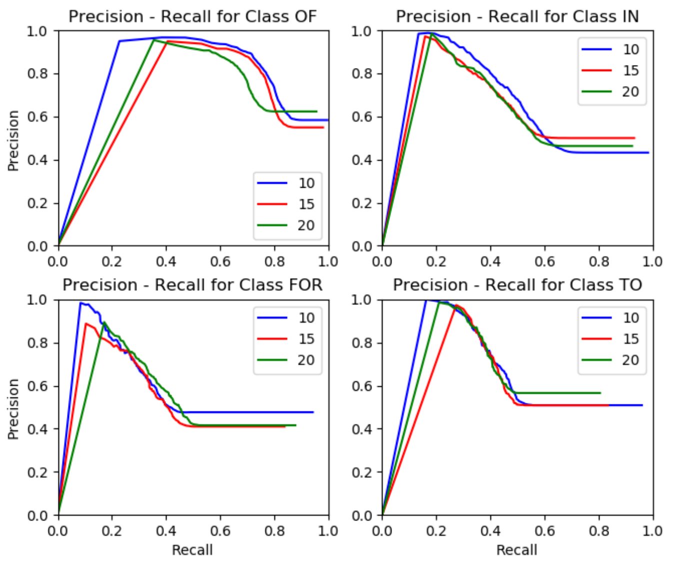

Figure 66 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the maximum length of the input sequence

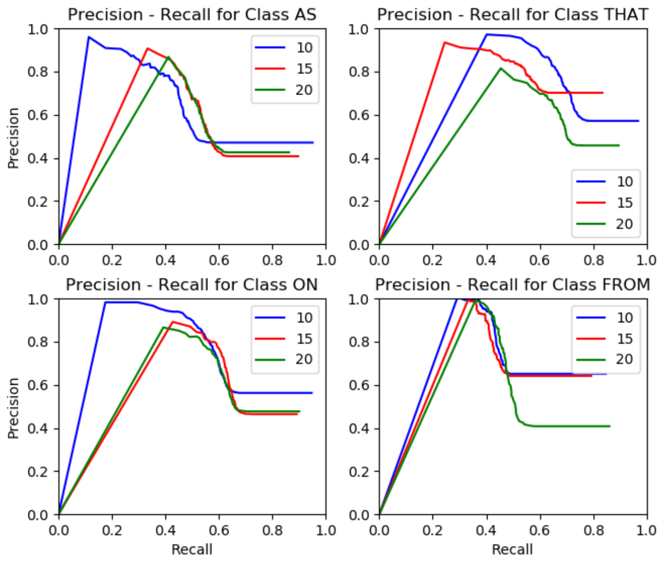

Figure 67 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the maximum length of the input sequence

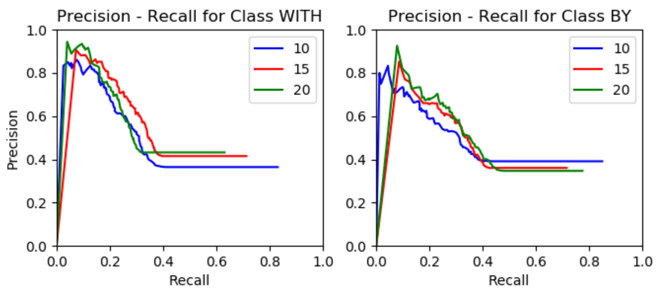

Figure 68 Precision vs. Recall for classes WITH (left) and BY (right) varying the maximum length of the input sequence

4.3.1.2.2 ROC Graphs

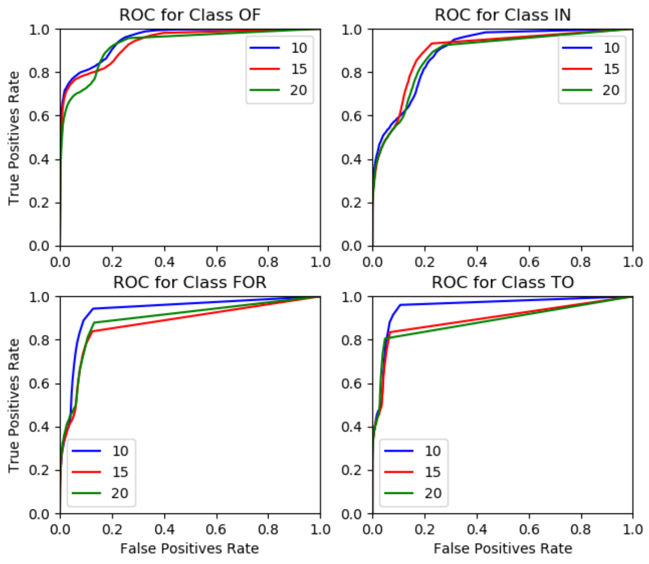

Figure 69 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the maximum length of the input sequence

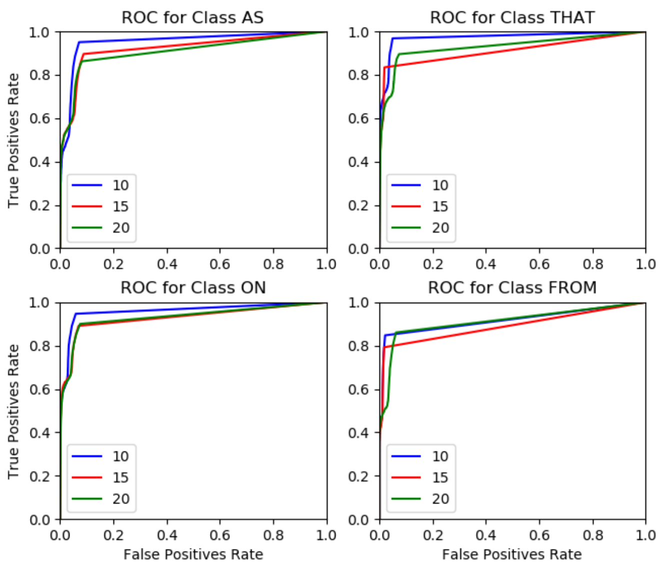

Figure 70 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the maximum length of the input sequence

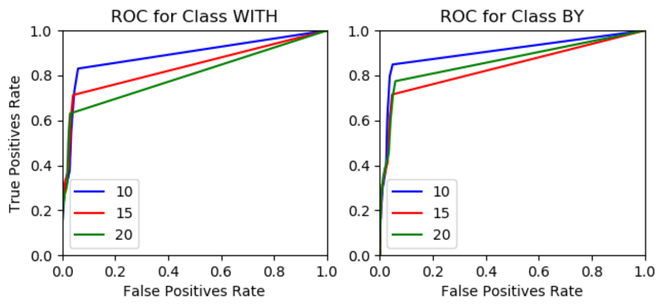

Figure 71 ROC for classes WITH (left) and BY (right) varying the maximum length of the input sequence

4.3.1.2.3 Metrics for Error Correction against a Threshold

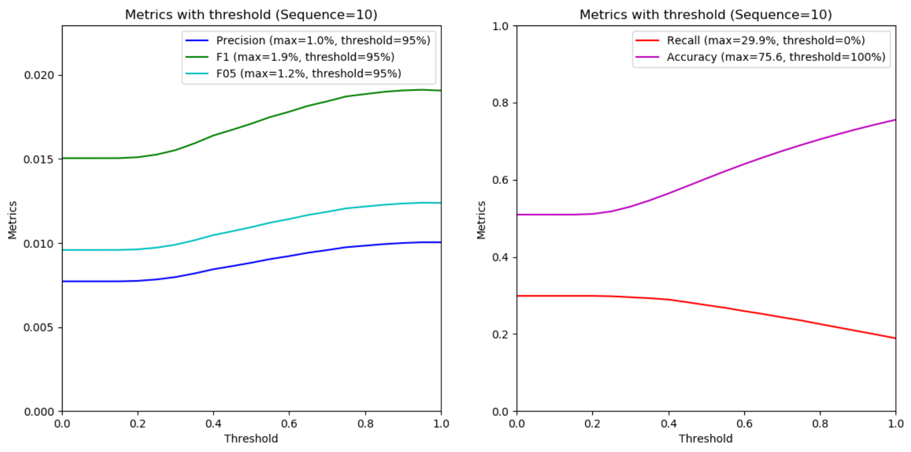

Figure 72 Correction Metrics against a Threshold at a maximum input sequence of 10

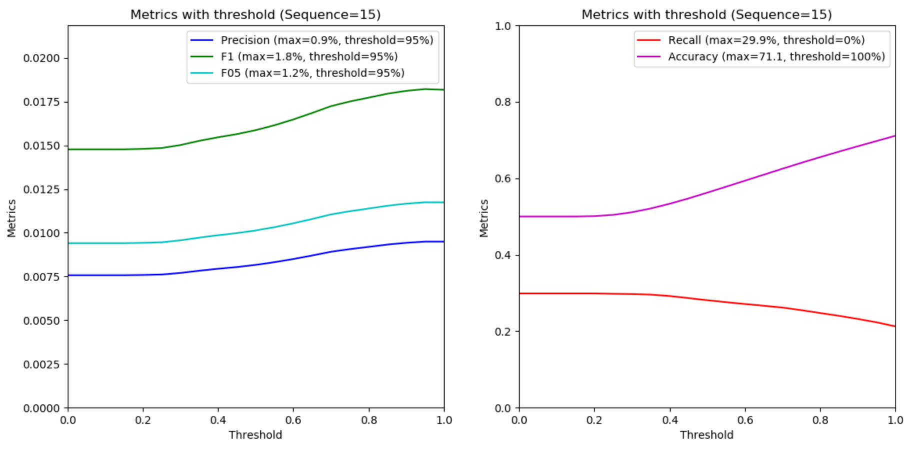

Figure 73 Correction Metrics against a Threshold at a maximum input sequence of 15

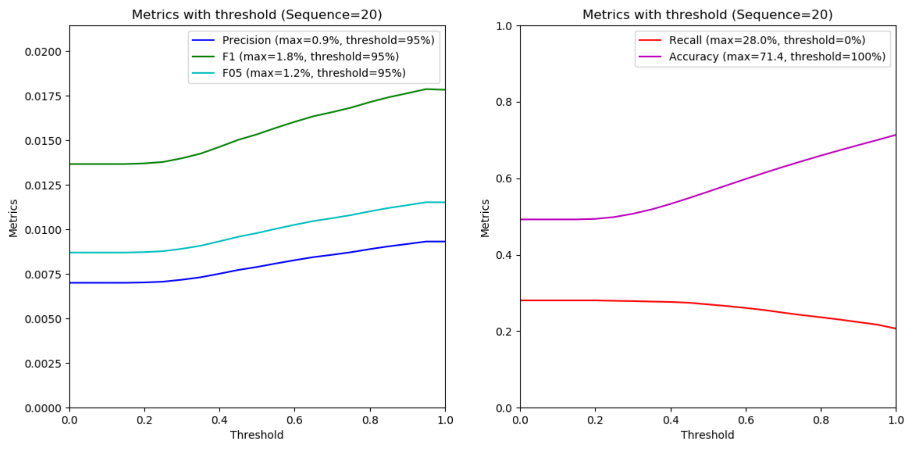

Figure 74 Correction Metrics against a Threshold at a maximum input sequence of 20

4.3.1.2.4 Metrics for Detection and Correction Tasks

Sequence	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
5	1.10%	37.8%	2.14%	1.37%	56.64%
10	0.77%	29.88%	1.50%	0.96%	50.94%
15	0.75%	29.88%	1.48%	0.94%	50.00%
20	0.70%	28.05%	1.37%	0.87%	49.24%

Table 22 Error correction for all classes varying the maximum length of the input sequence

Sequence	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
5	2.4%	82.32%	4.66%	2.98%	57.53%
10	2.06%	79.88%	4.022%	2.56%	51.89%
15	2.00%	79.27%	3.92%	2.49%	50.93%
20	1.94%	79.27%	3.86%	2.46%	50.20%

Table 23 Error detection for all classes varying the maximum length of the input sequence

Sequence	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
5	1.73%	45.25%	3.33%	2.14%	67.27%
10	1.14%	35.77%	2.20%	1.41%	60.50%
15	1.10%	35.77%	2.14%	1.37%	59.39%
20	1.02%	33.58%	1.97%	1.26%	58.48%

Table 24 Error correction for 10 top classes varying the maximum length of the input sequence

Sequence	Precision	Recall	F1	F05	Accuracy
5	3.34%	87.59%	6.44%	4.13%	68.15%
10	2.64%	83.21%	5.12%	3.28%	61.45%
15	2.50%	81.02%	4.96%	3.11%	60.29%
20	2.45%	81.02%	4.76%	3.04%	59.42%

Table 25 Error correction for 10 top classes varying the maximum length of the input sequence

4.4 Features

4.4.1 Tags

To test the effectiveness of adding POS tags to the input sequence (keeping the lexical features), the tag flag was set at True and the length of the embedding size was set at values of 10, 15, 20 and 25 while keeping all other parameters at their default value.

4.4.1.1 Training Phase

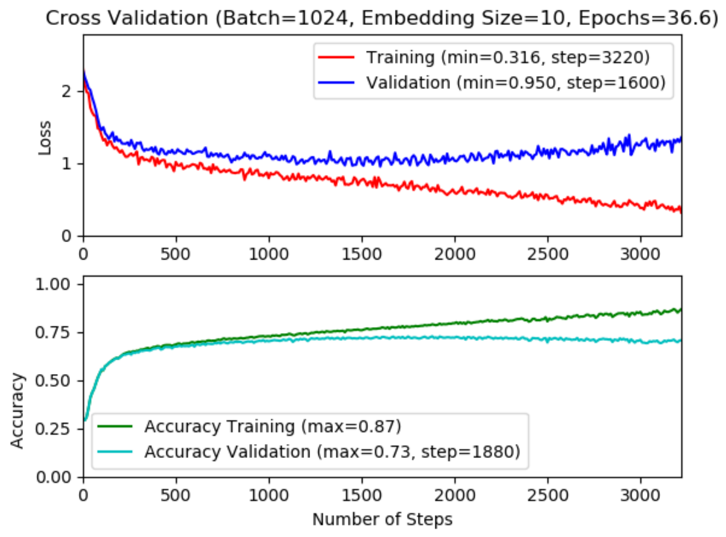

Figure 75 Cross validation adding embedded POS tags at a vector of size 10

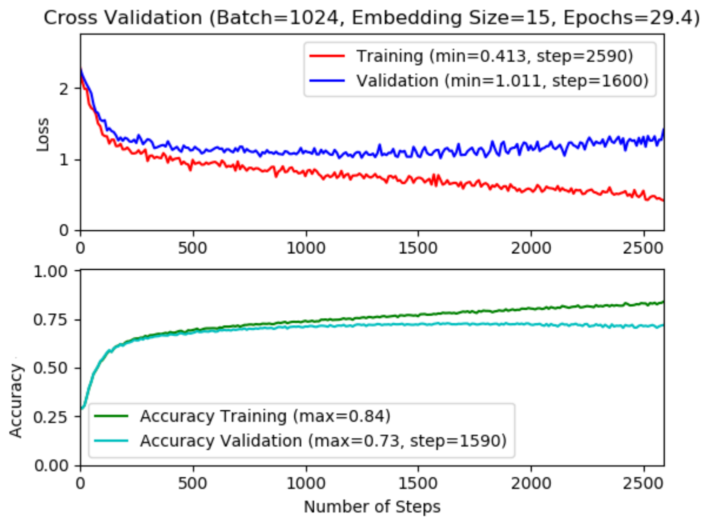

Figure 76 Cross validation adding embedded POS tags at a vector of size 15

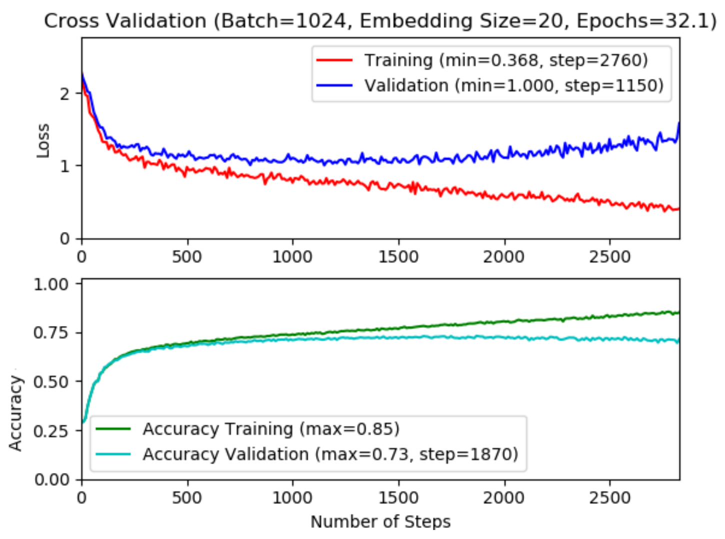

Figure 77 Cross validation adding embedded POS tags at a vector of size 20

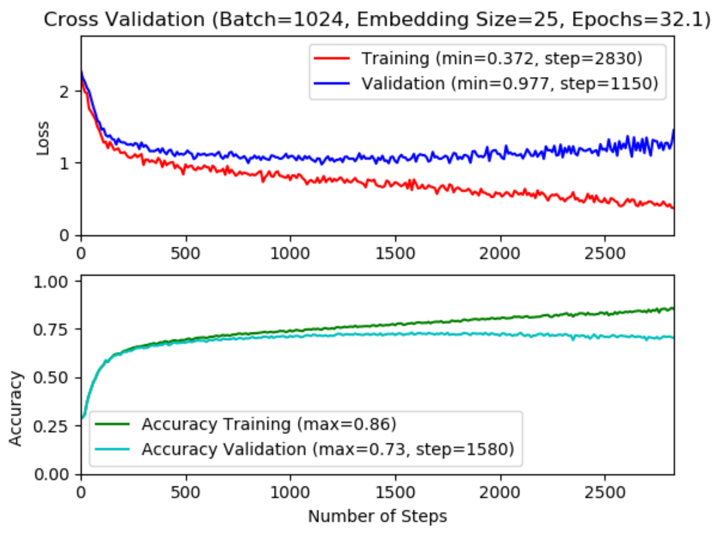

Figure 78 Cross validation adding embedded POS tags at a vector of size 25

4.4.1.2 Testing Phase

4.4.1.2.1 Precision-Recall Graphs

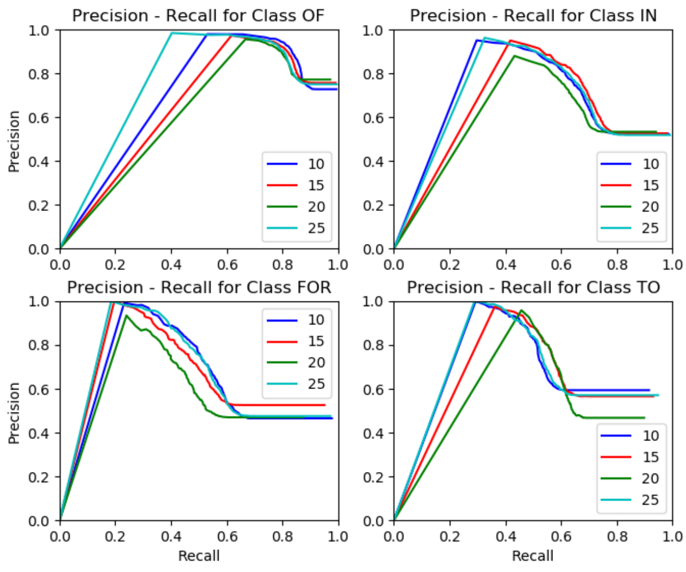

Figure 79 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) adding POS tags at different embedding sizes

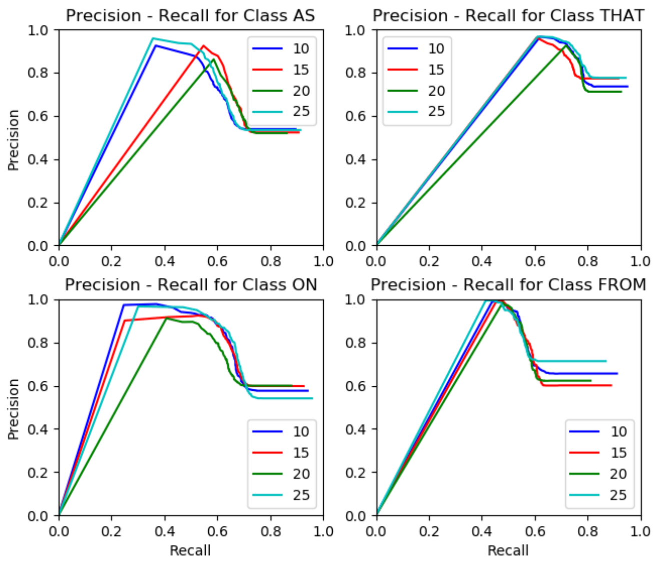

Figure 80 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) adding POS tags at different embedding sizes

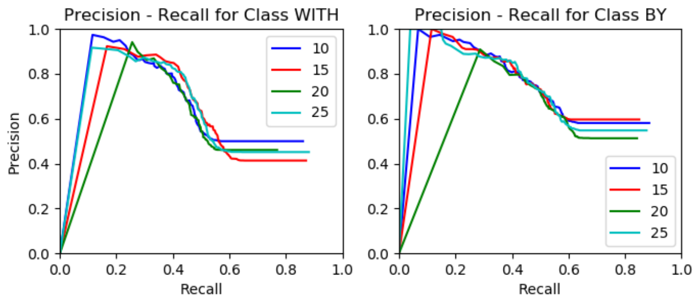

Figure 81 Precision vs. Recall for classes WITH (left) and BY (right) adding POS tags at different embedding sizes

4.4.1.2.2 ROC Graphs

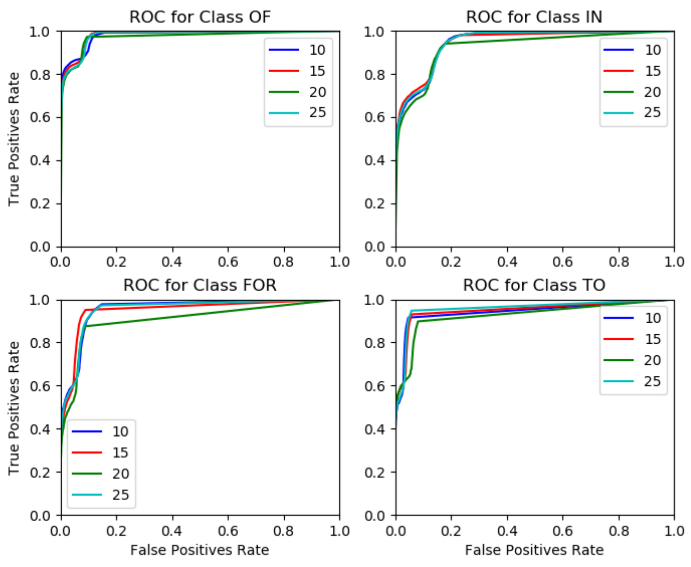

Figure 82 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) adding POS tags at different embedding sizes

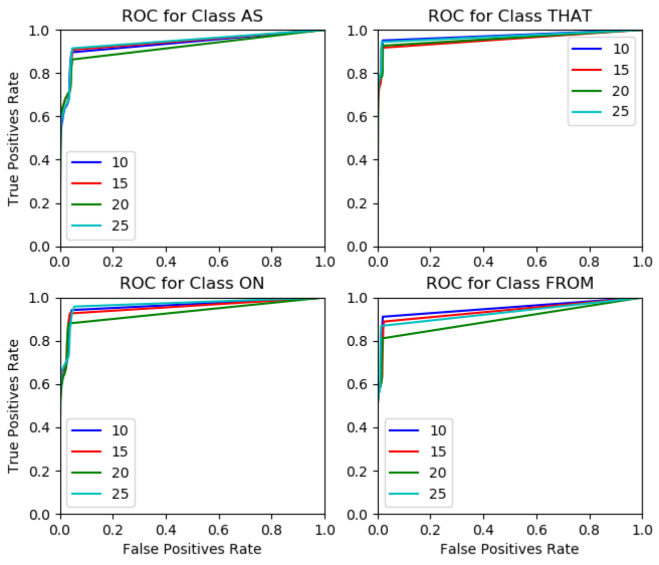

Figure 83 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) adding POS tags at different embedding sizes

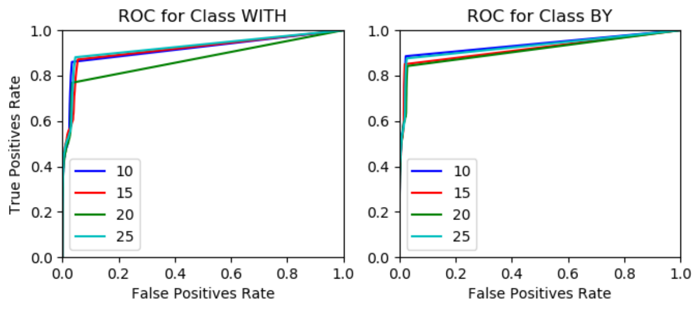

Figure 84 ROC for classes WITH (left) and BY (right) adding POS tags at different embedding sizes

4.4.1.2.3 Metrics for Error Correction with Threshold

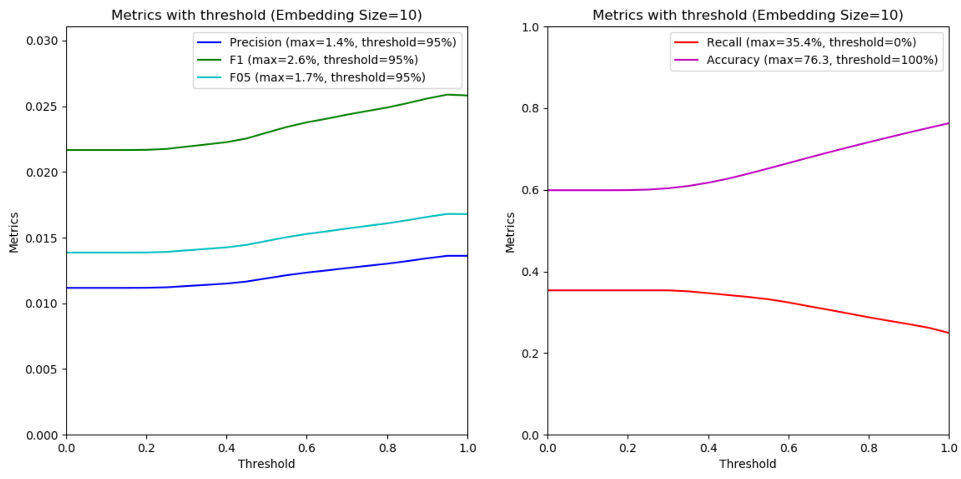

Figure 85 Correction Metrics against a Threshold adding embedded POS tags at a vector of size 10

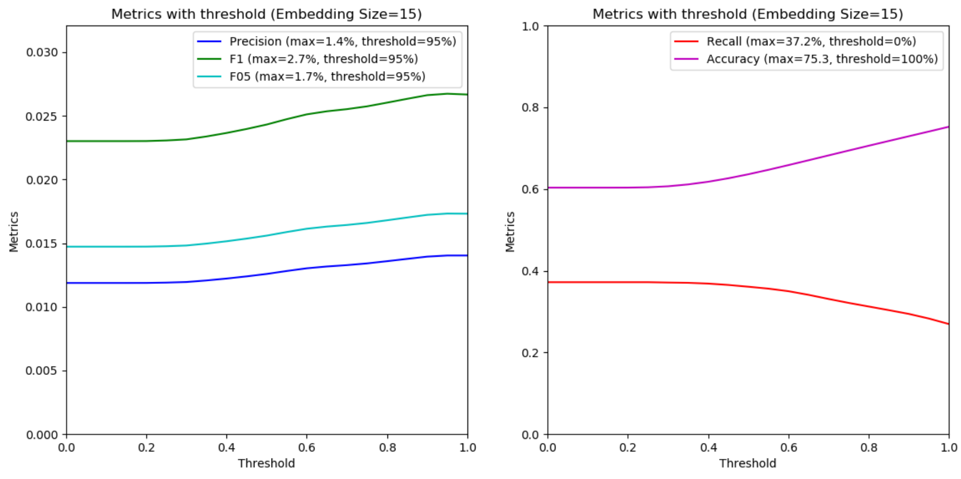

Figure 86 Correction Metrics against a Threshold adding embedded POS tags at a vector of size 15

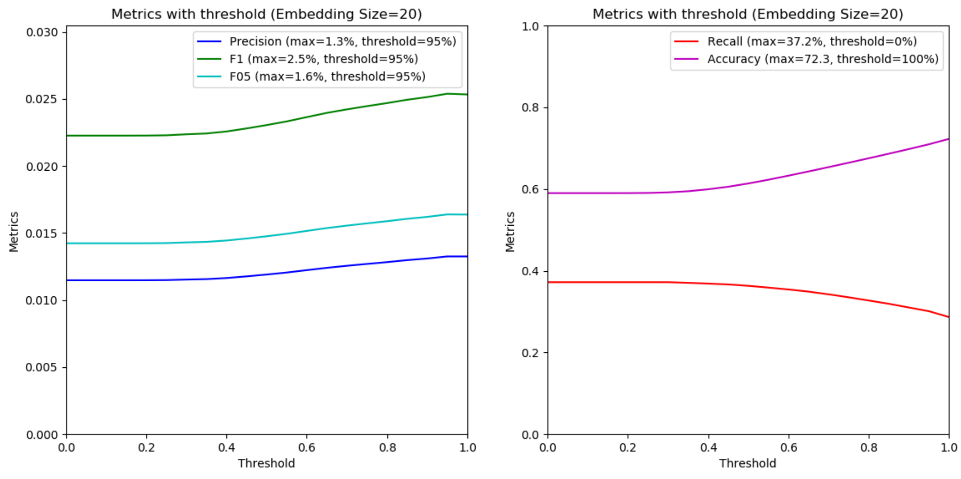

Figure 87 Correction Metrics against a Threshold adding embedded POS tags at a vector of size 20

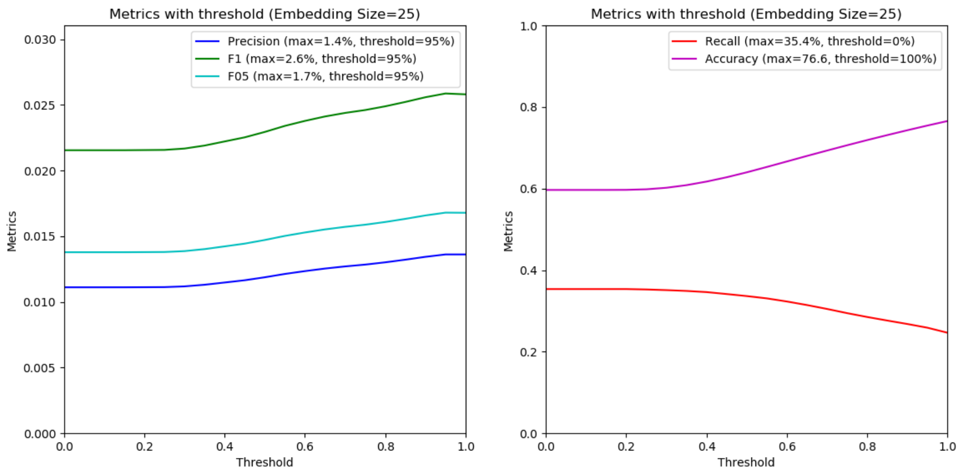

Figure 88 Correction Metrics against a Threshold adding embedded POS tags at a vector of size 25

4.4.1.2.4 Metrics for Correction and Detection Tasks

Embedding Tag	Precision	Recall	F1	F05	Accuracy
Without tags	1.10%	37.8%	2.14%	1.37%	56.64%
10	1.11%	35.36%	2.17%	1.38%	59.90%
15	1.18%	37.19%	2.30%	1.47%	60.34%
20	1.15%	37.19%	2.23%	1.42%	59.00%
25	1.11%	35.36%	2.15%	1.38%	59.67%

Table 26 Error correction for all classes adding POS tags at different embedding sizes

Embedding Tag	Precision	Recall	F1	F05	Accuracy
Without tags	2.4%	82.32%	4.66%	2.98%	57.53%
10	2.45%	77.44%	4.74%	3.03%	60.75%
15	2.43%	76.21%	4.72%	3.02%	61.13%
20	2.48%	80.49%	4.82%	3.08%	59.87%
25	2.43%	77.43%	4.71%	3.02%	60.52%

Table 27 Error detection for all classes adding POS tags at different embedding sizes

Embedding Tag	Precision	Recall	F1	F05	Accuracy
Without tags	1.73%	45.25%	3.33%	2.14%	67.27%
10	1.84%	42.33%	3.52%	2.27%	71.13%
15	1.97%	44.52%	3.77%	2.43%	71.66%
20	1.86%	44.52%	3.57%	2.30%	70.08%
25	1.82%	42.33%	3.49%	2.25%	70.87%

Table 28 Error correction for 10 top classes adding POS tags at different embedding sizes

Embedding Tag	Precision	Recall	F1	F05	Accuracy
Without Tags	3.34%	87.59%	6.44%	4.13%	68.15%
10	3.55%	81.75%	6.81%	4.39%	71.98%
15	3.55%	80.29%	6.79%	4.38%	72.43%
20	3.48%	83.21%	6.68%	4.30%	70.90%
25	3.52%	81.75%	6.74%	4.35%	71.71%

Table 29 Error correction for 10 top classes adding POS tags at different embedding sizes

4.4.2 Parse Features

To test the effectiveness of adding the parent index of the dependency tree to the input sequence (keeping the lexical features), the index flag was set at True and the length of the embedding size was set at values of 10, 15, 20 and 25 while keeping all other parameters at their default value.

4.4.2.1 Training Phase

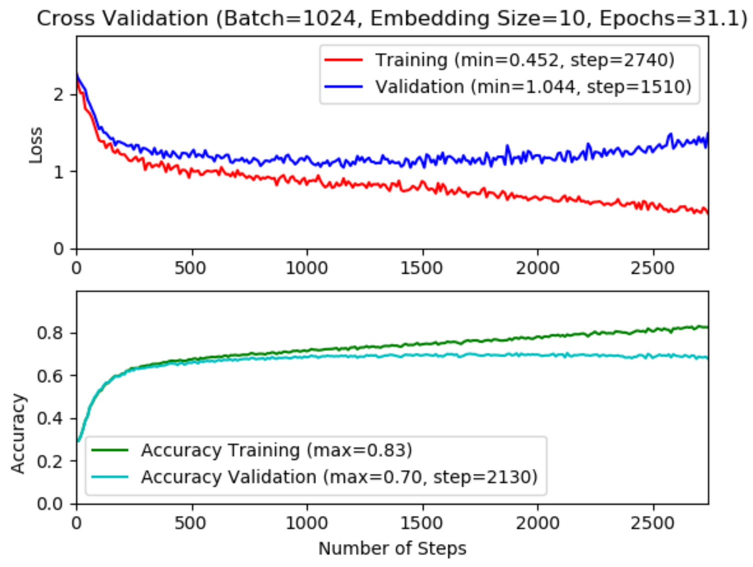

Figure 89 Cross validation adding embedded parent index of dependency tree at a vector of size 10

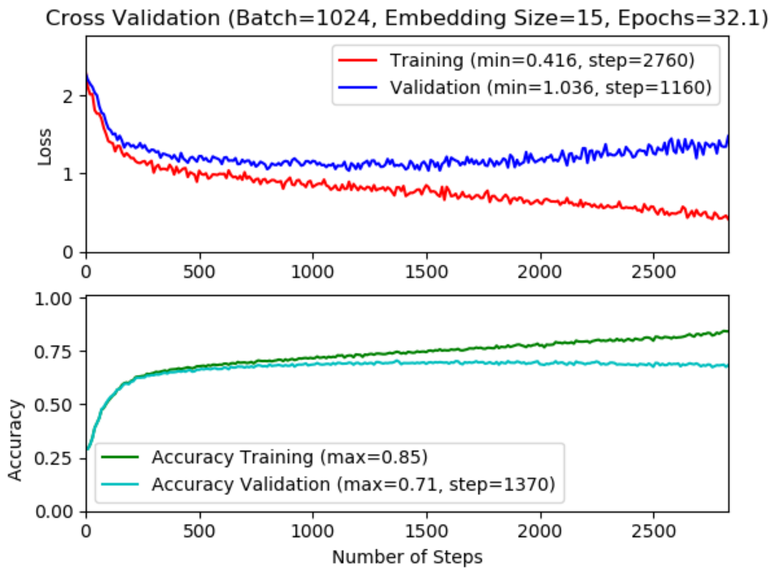

Figure 90 Cross validation adding embedded parent index of dependency tree at a vector of size 15

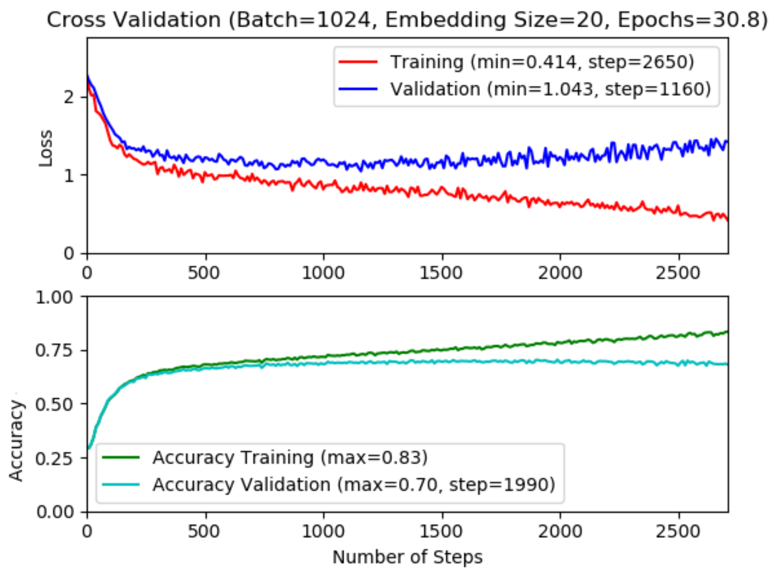

Figure 91 Cross validation adding embedded parent index of dependency tree at a vector of size 20

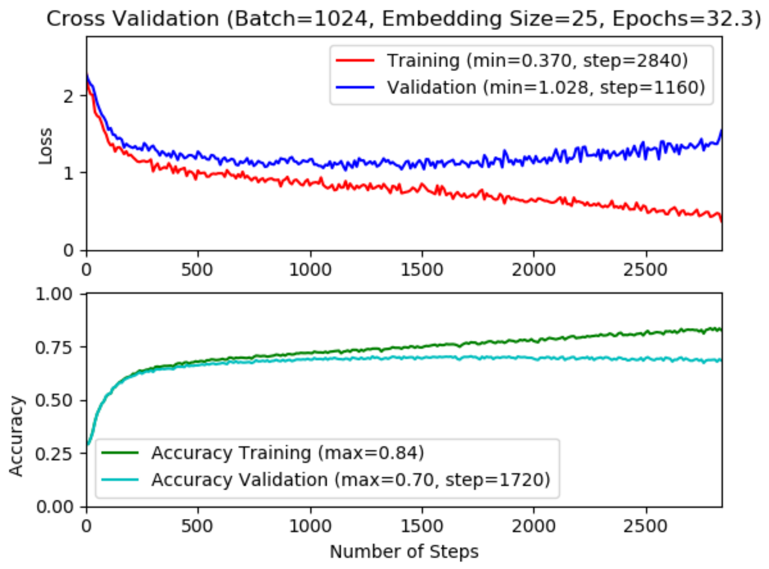

Figure 92 Cross validation adding embedded parent index of dependency tree at a vector of size 25

4.4.2.2 Testing Phase

4.4.2.2.1 Precision-Recall Graphs

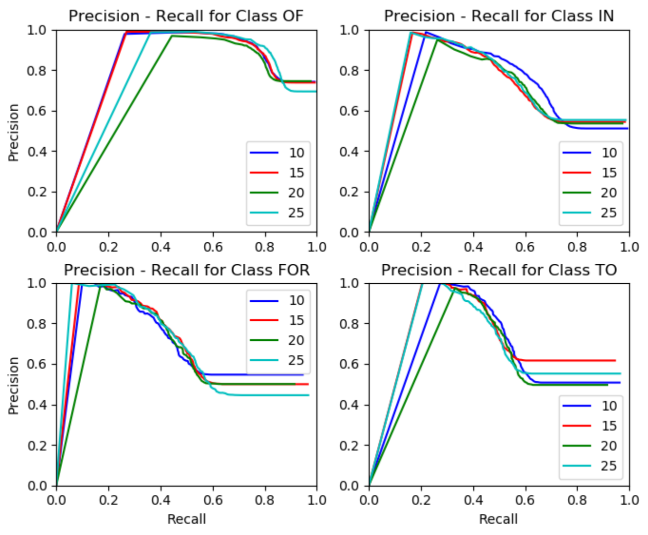

Figure 93 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) adding parent index of dependency tree at different embedding sizes

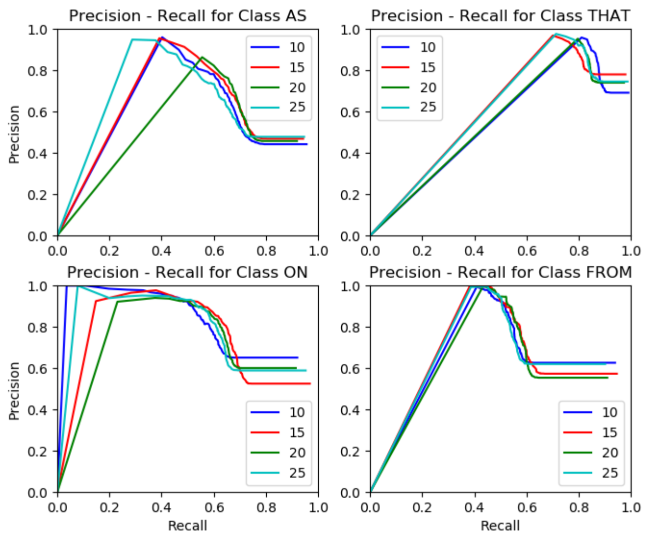

Figure 94 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) adding parent index of dependency tree at different embedding sizes

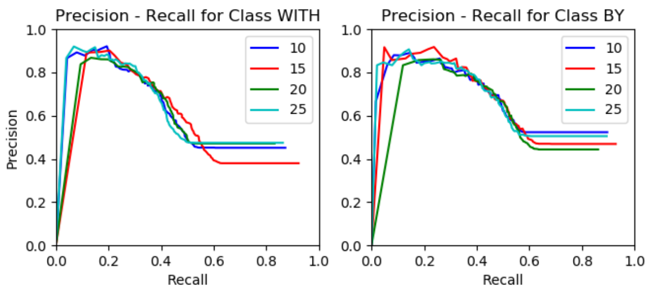

Figure 95 Precision vs. Recall for classes WITH (left) and BY (right) adding parent index of dependency tree at different embedding sizes

4.4.2.2.2 ROC Graphs

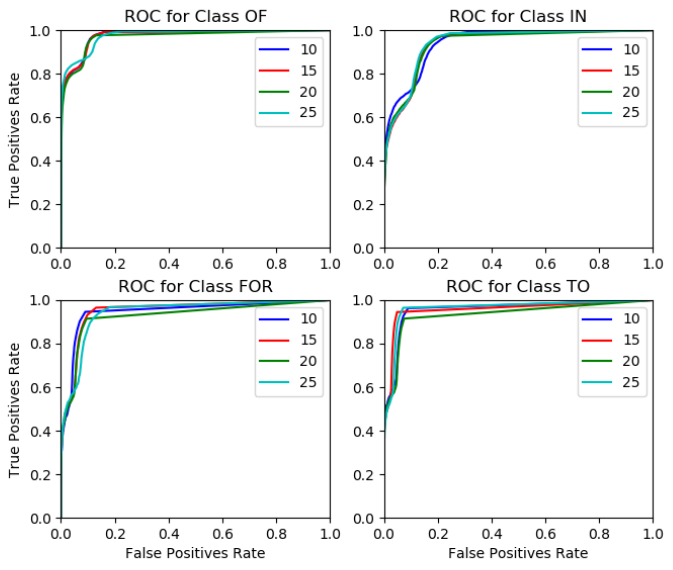

Figure 96 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) adding parent index of dependency tree at different embedding sizes

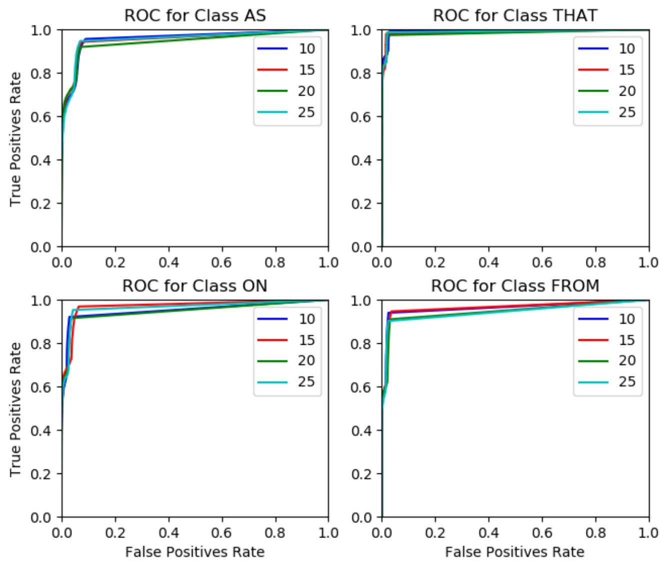

Figure 97 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) adding parent index of dependency tree at different embedding sizes

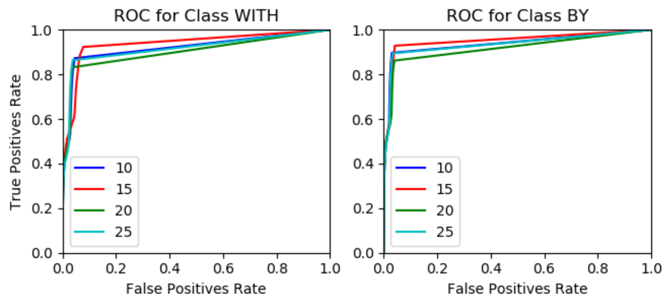

Figure 98 ROC for classes WITH (left) and BY (right) adding parent index of dependency tree at different embedding sizes

4.4.2.2.3 Metrics for Error Correction against Threshold

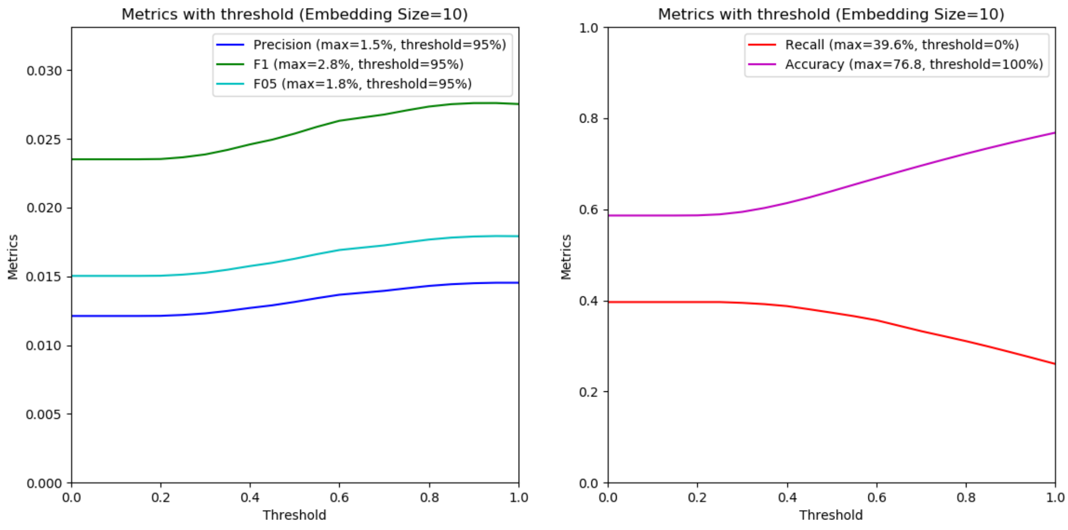

Figure 99 Correction Metrics against a Threshold adding embedded parent index of dependency tree at a vector of size 10

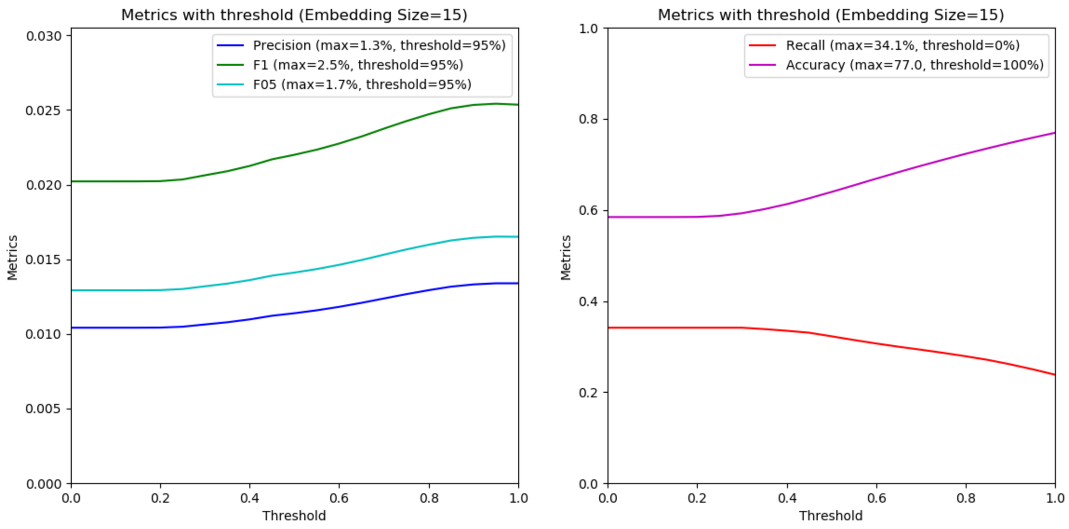

Figure 100 Correction Metrics against a Threshold adding embedded parent index of dependency tree at a vector of size 15

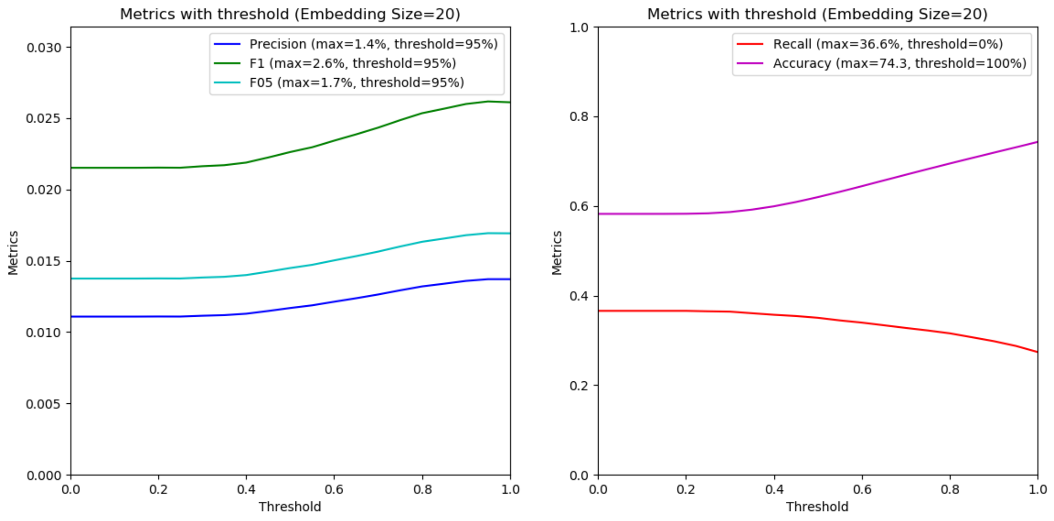

Figure 101 Correction Metrics against a Threshold adding embedded parent index of dependency tree at a vector of size 20

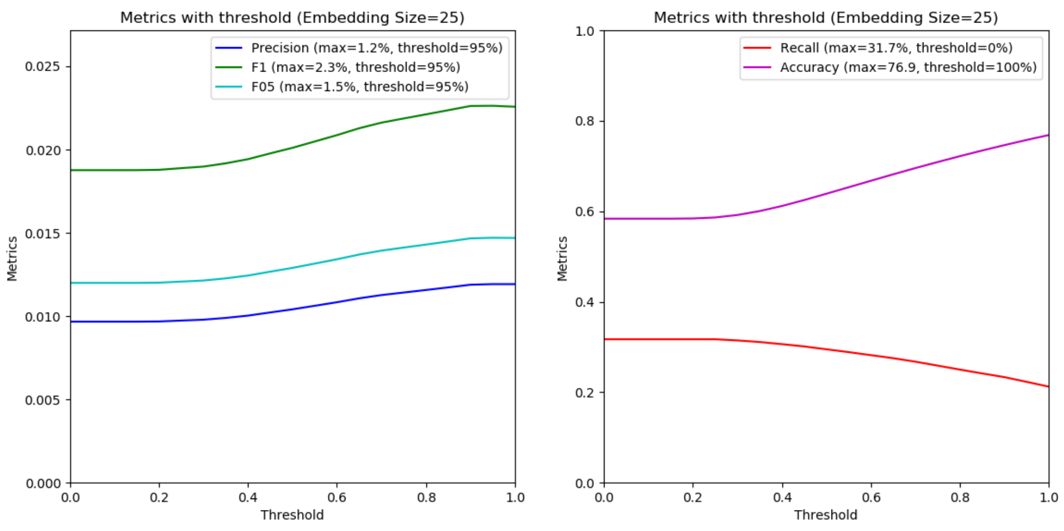

Figure 102 Correction Metrics against a Threshold adding embedded parent index of dependency tree at a vector of size 25

4.4.2.2.4 Metrics for Error Correction and Detection Tasks

Embedding Index	Precision	Recall	F1	F05	Accuracy
Without Index	1.10%	37.8%	2.14%	1.37%	56.64%
10	1.21%	39.63%	2.35%	1.50%	58.64%
15	1.04%	34.14%	2.02%	1.29%	58.45%
20	1.10%	36.58%	2.15%	1.37%	58.23%
25	0.96%	31.70%	1.88%	1.20%	58.37%

Table 30 Error correction for all classes adding parent index of dependency tree at different embedding sizes

Embedding Index	Precision	Recall	F1	F05	Accuracy
Without Index	2.4%	82.32%	4.66%	2.98%	57.53%
10	2.34%	76.82%	4.55%	2.91%	59.38%
15	2.34%	76.82%	4.54%	2.90%	59.30%
20	2.34%	77.44%	4.55%	2.91%	59.04%
25	2.27%	74.39%	4.40%	2.81%	59.22%

Table 31 Error detection for all classes adding parent index of dependency tree at different embedding sizes

Embedding Index	Precision	Recall	F1	F05	Accuracy
Without Index	1.73%	45.25%	3.33%	2.14%	67.27%
10	1.95%	47.44%	3.74%	2.41%	69.63%
15	1.67%	40.87%	3.22%	2.07%	69.41%
20	1.77%	43.79%	3.41%	2.20%	69.14%
25	1.55%	37.96%	2.98%	1.92%	69.32%

Table 32 Error correction for 10 top classes adding parent index of dependency tree at different embedding sizes

Embedding Index	Precision	Recall	F1	F05	Accuracy
Without Index	3.34%	87.59%	6.44%	4.13%	68.15%
10	3.39%	82.48%	6.51%	4.20%	70.37%
15	3.26%	79.56%	6.27%	4.03%	70.24%
20	3.37%	83.21%	6.49%	4.18%	69.97%
25	3.17%	77.37%	6.09%	3.92%	70.15%

Table 33 Error detection for 10 top classes adding parent index of dependency tree at different embedding sizes

4.4.3 Limiting Minimum Number of Lexical Features

To test the effectiveness of setting a lower limit to the number of lexical features according to their number of occurrences, the parameter lower limit was set at values of 1, 2, 4, 8 and 16 for each experiment, while keeping all other parameters at their default value.

4.4.3.1 Training Phase

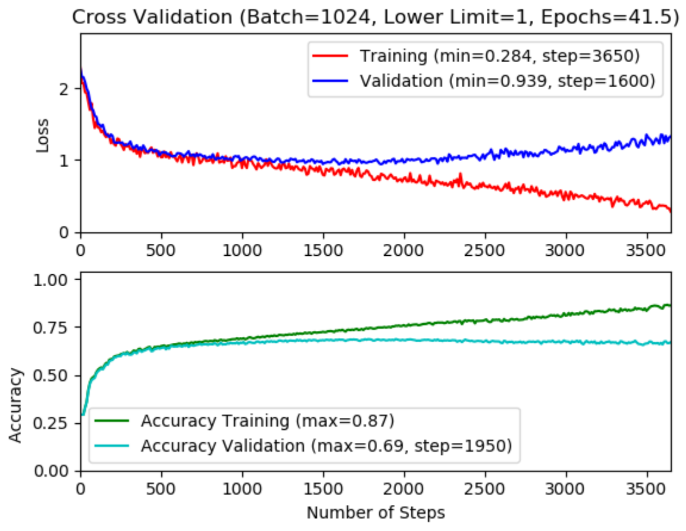

Figure 103 Cross validation limiting the minimum number of lexical occurrences at more than 1

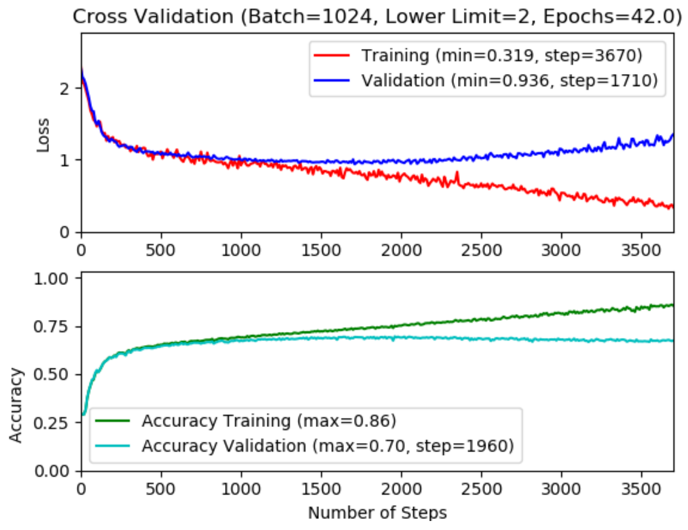

Figure 104 Cross validation limiting the minimum number of lexical occurrences at more than 2

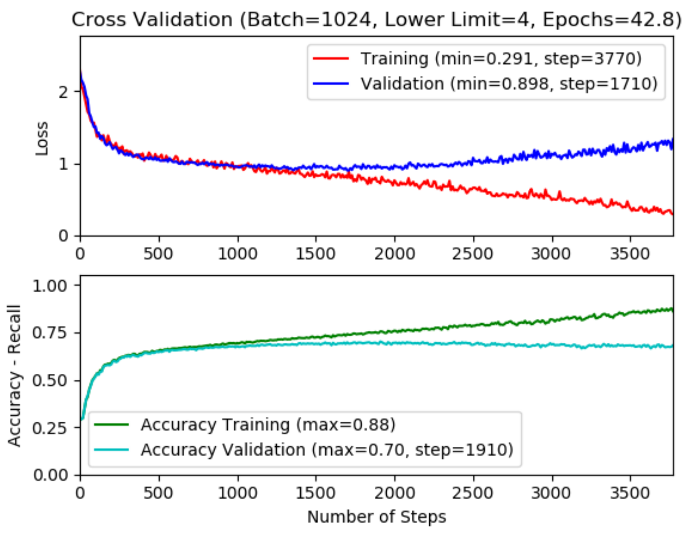

Figure 105 Cross validation limiting the minimum number of lexical occurrences at more than 4

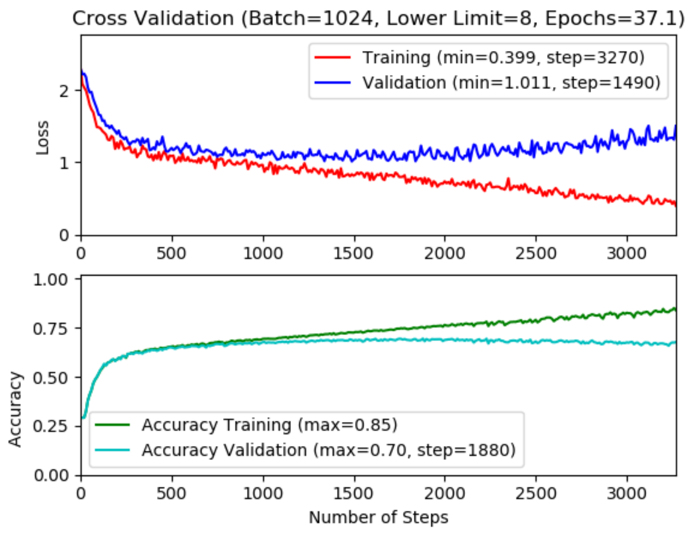

Figure 106 Cross validation limiting the minimum number of lexical occurrences at more than 8

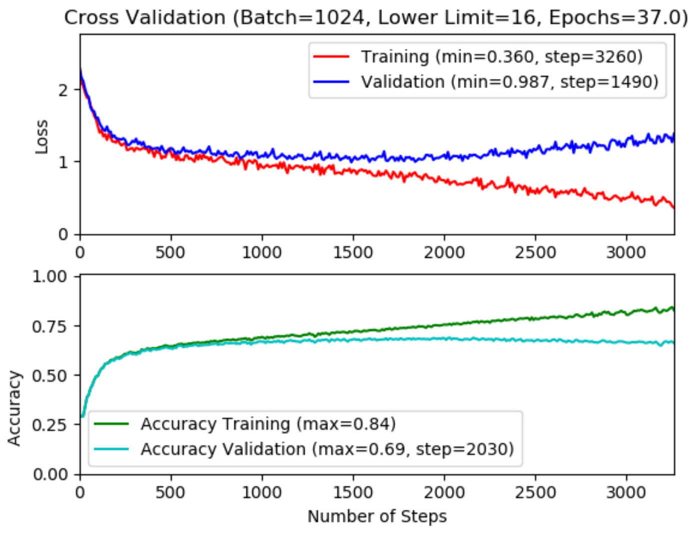

Figure 107 Cross validation limiting the minimum number of lexical occurrences at more than 16

4.4.3.2 Testing Phase

4.4.3.2.1 Precision-Recall Graphs

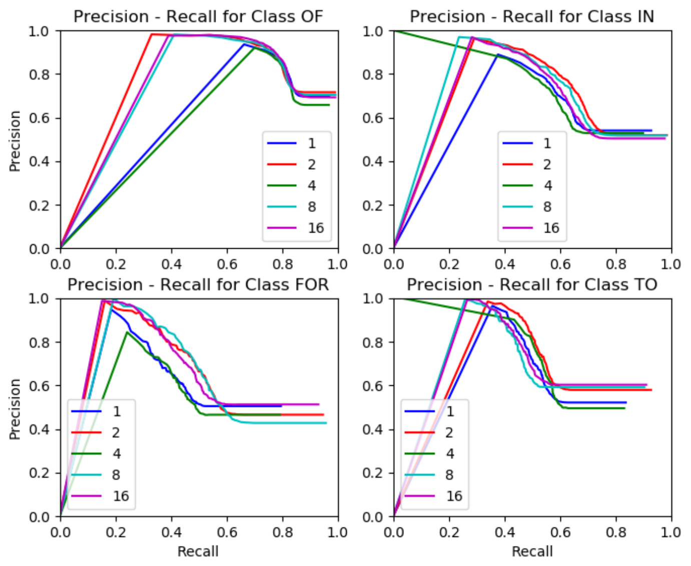

Figure 108 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the minimum number of lexical occurrences

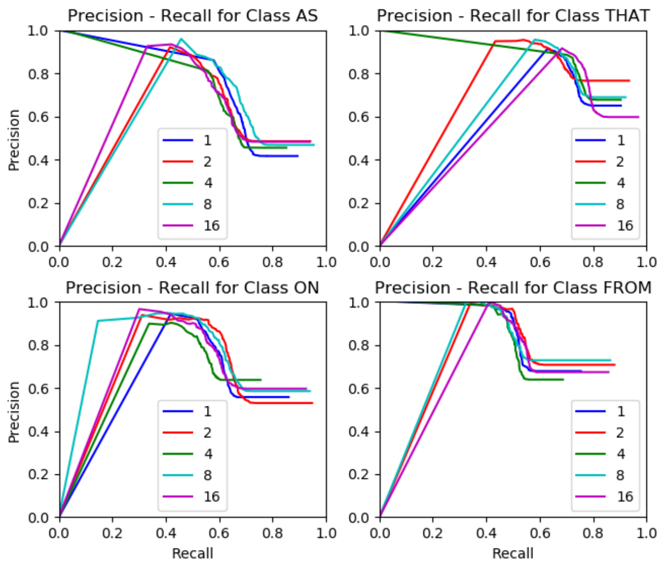

Figure 109 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the minimum number of lexical occurrences

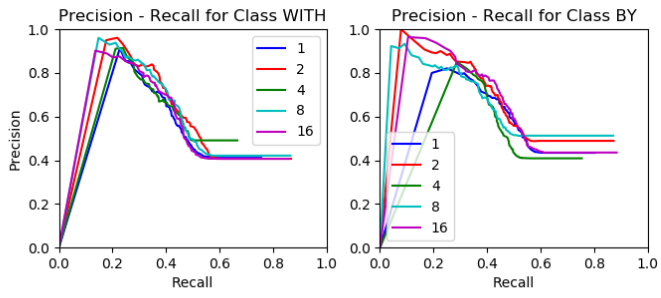

Figure 110 Precision vs. Recall for classes WITH (left) and BY (right) varying the minimum number of lexical occurrences

4.4.3.2.2 ROC Graphs

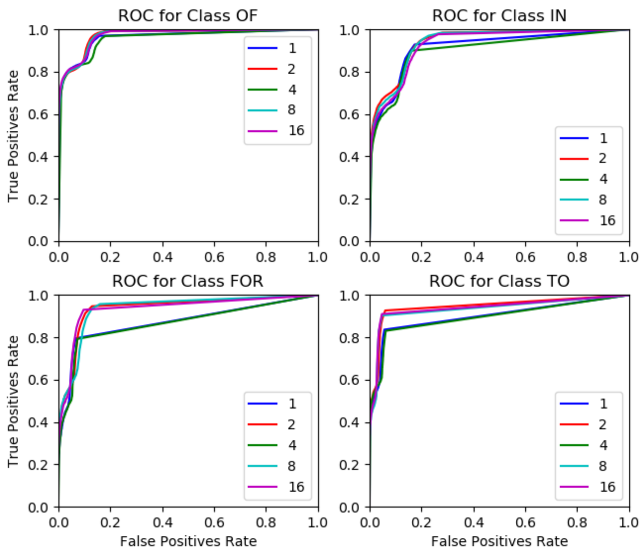

Figure 111 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the minimum number of lexical occurrences

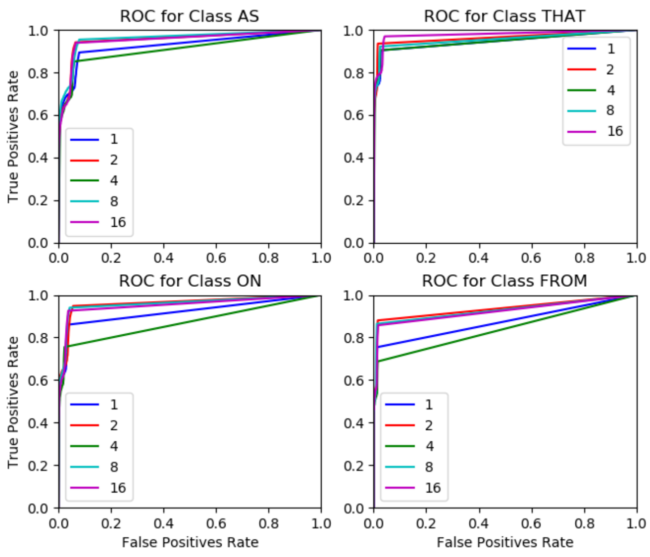

Figure 112 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the minimum number of lexical occurrences

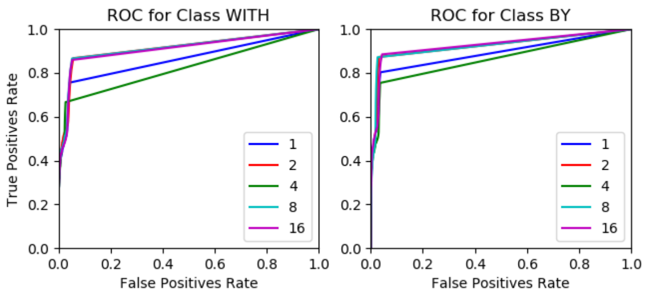

Figure 113 ROC for classes WITH (left) and BY (right) varying the minimum number of lexical occurrences

4.4.3.2.3 Metrics for Error Correction against a Threshold

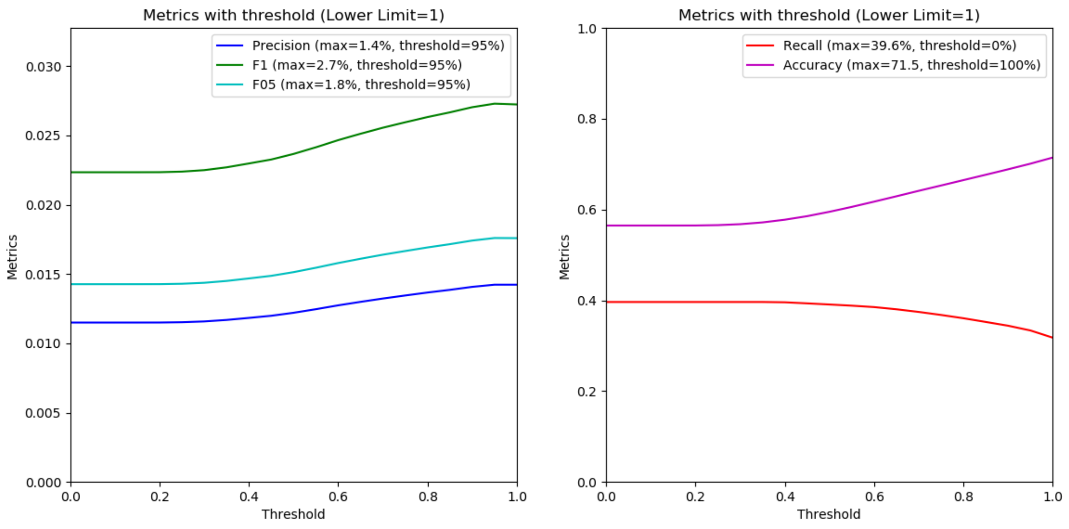

Figure 114 Error Correction Metrics against a Threshold limiting the minimum number of lexical occurrences at more than 1

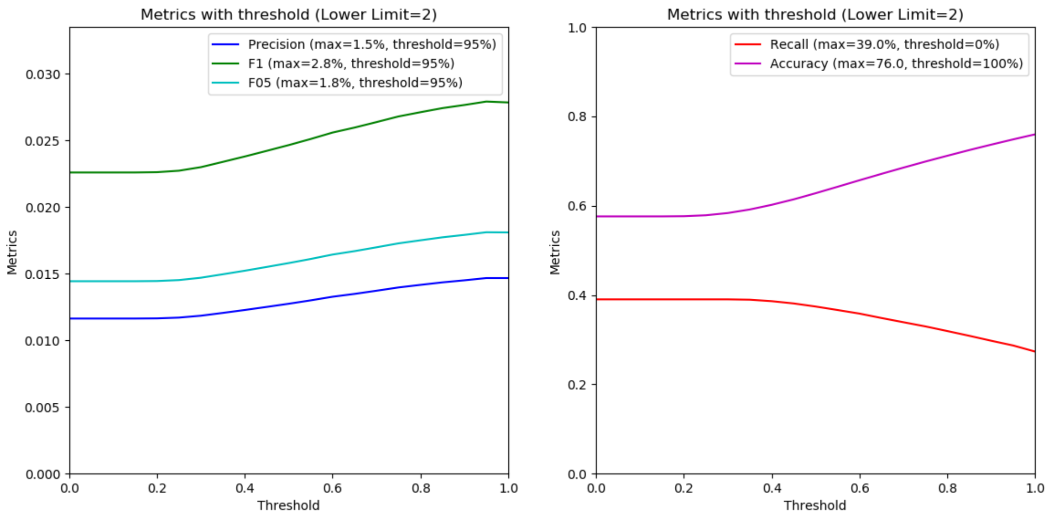

Figure 115 Error Correction Metrics against a Threshold limiting the minimum number of lexical occurrences at more than 2

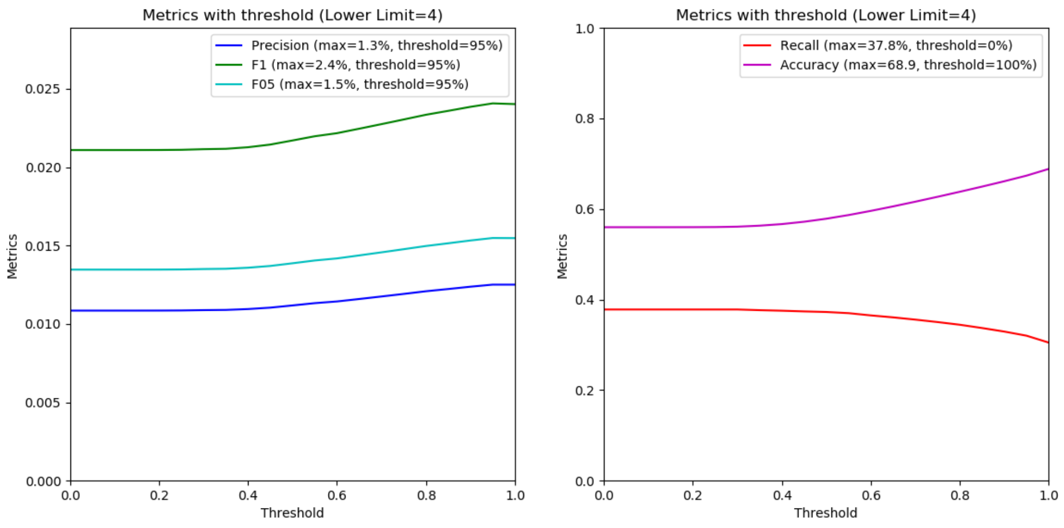

Figure 116 Error Correction Metrics against a Threshold limiting the minimum number of lexical occurrences at more than 4

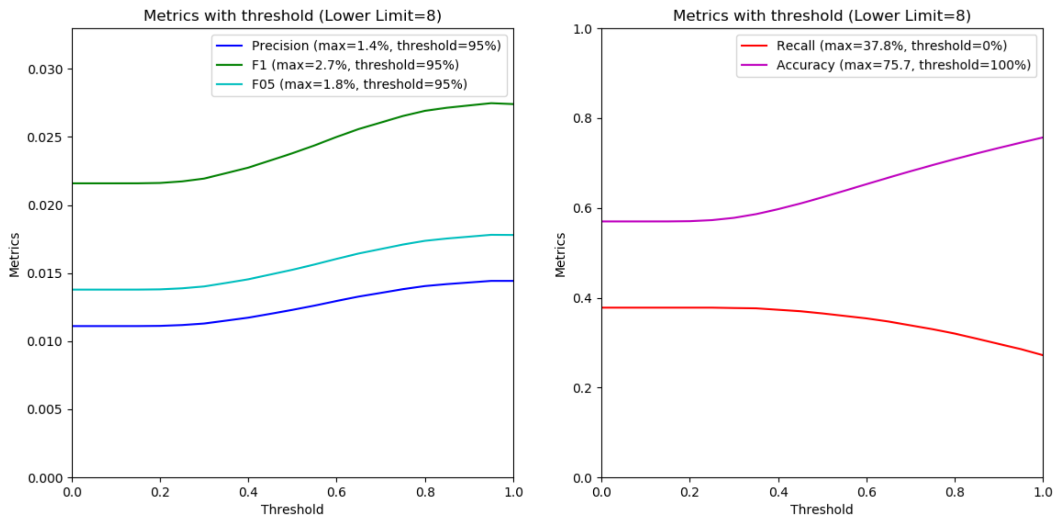

Figure 117 Error Correction Metrics against a Threshold limiting the minimum number of lexical occurrences at more than 8

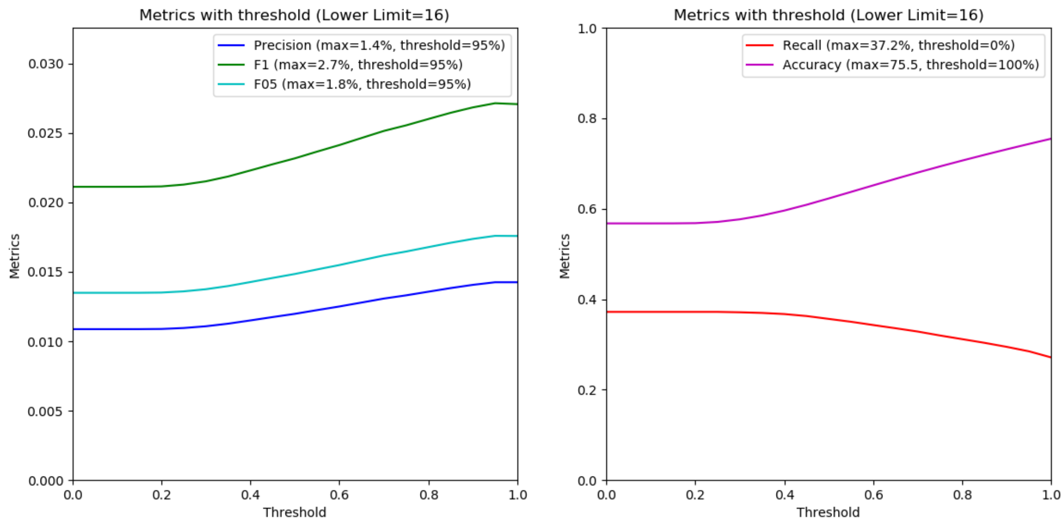

Figure 118 Error Correction Metrics against a Threshold limiting the minimum number of lexical occurrences at more than 16

4.4.3.2.4 Metrics for Error Correction and Detection Tasks

Lower Limit	Precision	Recall	F1	F05	Accuracy
0	1.10%	37.8%	2.14%	1.37%	56.64%
1	1.15%	39.63%	2.23%	1.43%	56.48%
2	1.16%	39.02%	2.26%	1.44%	57.61%
4	1.08%	37.80%	2.11%	1.35%	55.98%
8	1.11%	37.80%	2.16%	1.38%	57.00%
16	1.09%	37.19%	2.11%	1.35%	56.76%

Table 34 Error correction for all classes limiting the minimum number of lexical occurrences

Lower Limit	Precision	Recall	F1	F05	Accuracy
0	2.4%	82.32%	4.66%	2.98%	57.53%
1	2.48%	85.36%	4.81%	3.07%	57.38%
2	2.43%	81.70%	4.73%	3.02%	58.46%
4	2.45%	85.36%	4.76%	3.04%	56.91%
8	2.44%	82.93%	4.74%	3.02%	57.90%
16	2.51%	85.97%	4.88%	3.11%	57.73%

Table 35 Error detection for all classes limiting the minimum number of lexical occurrences

Lower Limit	Precision	Recall	F1	F05	Accuracy
0	1.73%	45.25%	3.33%	2.14%	67.27%
1	1.80%	47.44%	3.47%	2.23%	67.11%
2	1.84%	46.71%	3.55%	2.29%	68.43%
4	1.69%	45.25%	3.25%	2.09%	66.50%
8	1.75%	45.25%	3.37%	2.17%	67.71%
16	1.70%	44.52%	3.29%	2.11%	67.42%

Table 36 Error correction for 10 top classes limiting the minimum number of lexical occurrences

Lower Limit	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
0	3.34%	87.59%	6.44%	4.13%	68.15%
1	3.27%	86.13%	6.30%	4.05%	67.92%
2	3.35%	84.67%	6.44%	4.15%	69.23%
4	3.24%	86.86%	6.24%	4.01%	67.37%
8	3.30%	85.40%	6.36%	4.09%	68.55%
16	3.41%	89.05%	6.58%	4.23%	68.36%

Table 37 Error correction for 10 top classes limiting the minimum number of lexical occurrences

4.4.4 Original Preposition

The use of the original preposition was tested by setting the original parameter at values of *"include"* and *"replace"* for each experiment, while all other parameters kept their default values. The experiment with the original parameter at a value of *"exclude"* was already tested in the hidden cells section.

4.4.4.1 Training Phase

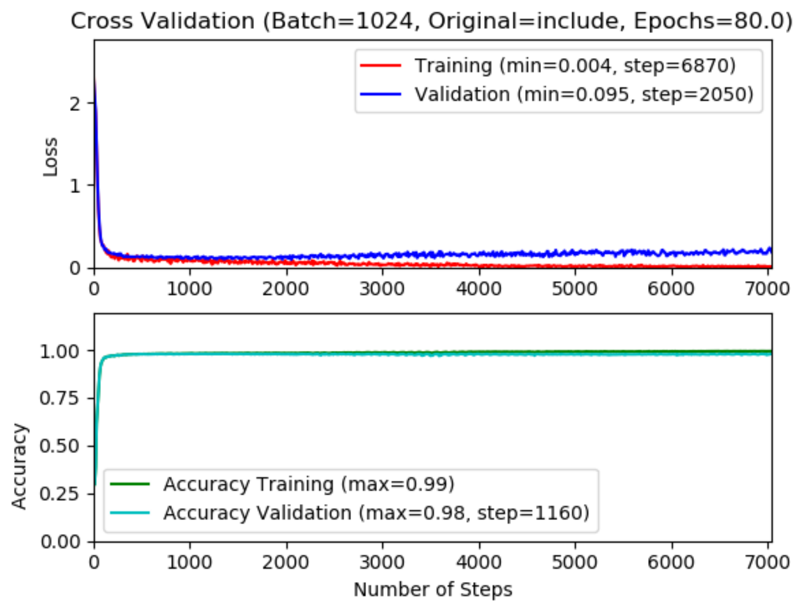

Figure 119 Cross validation including original preposition

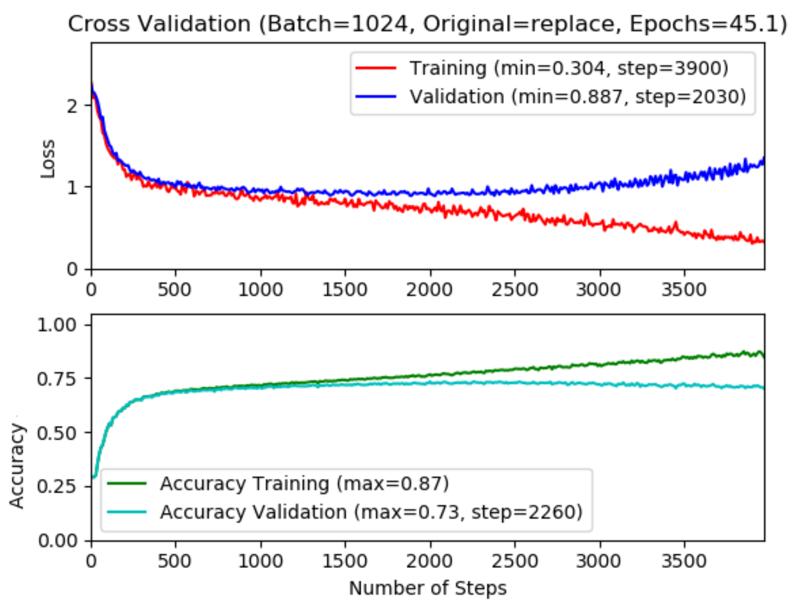

Figure 120 Cross validation replacing original preposition

4.4.4.2 Testing Phase

4.4.4.2.1 Precision-Recall Graphs

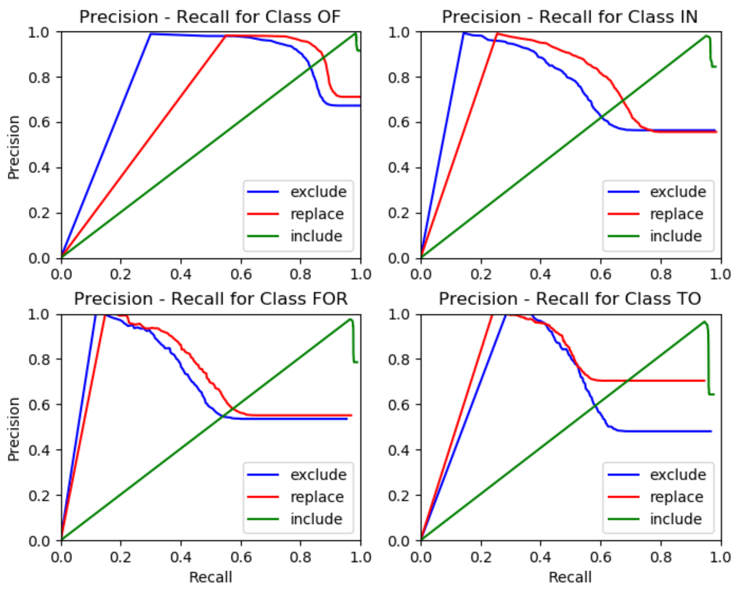

Figure 121 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the original preposition

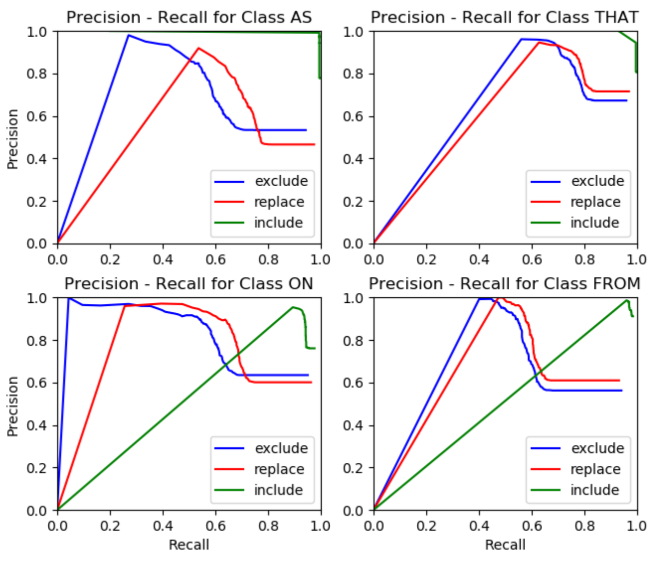

Figure 122 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the original preposition

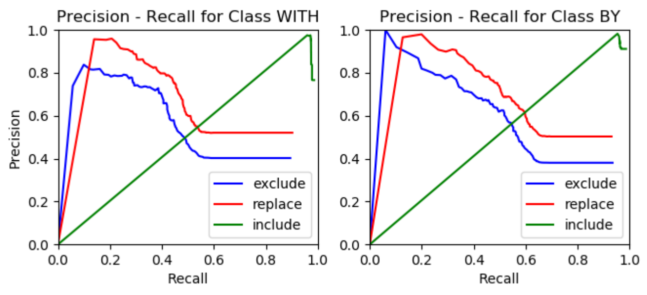

Figure 123 Precision vs. Recall for classes WITH (left) and BY (right) varying the original preposition

4.4.4.2.2 ROC Graphs

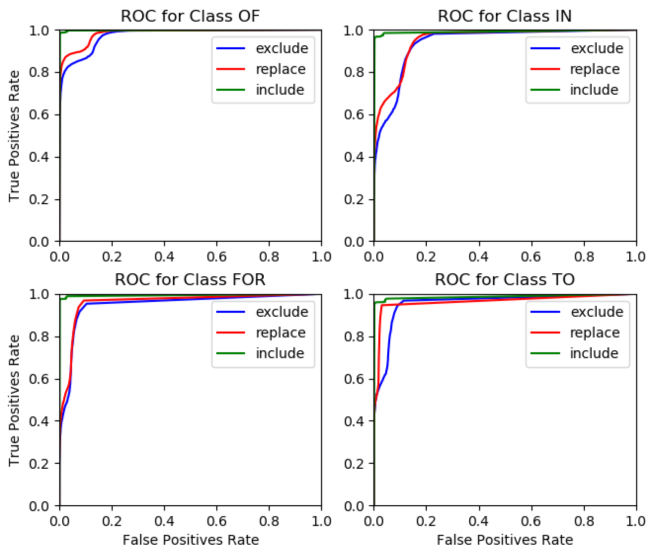

Figure 124 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the original preposition

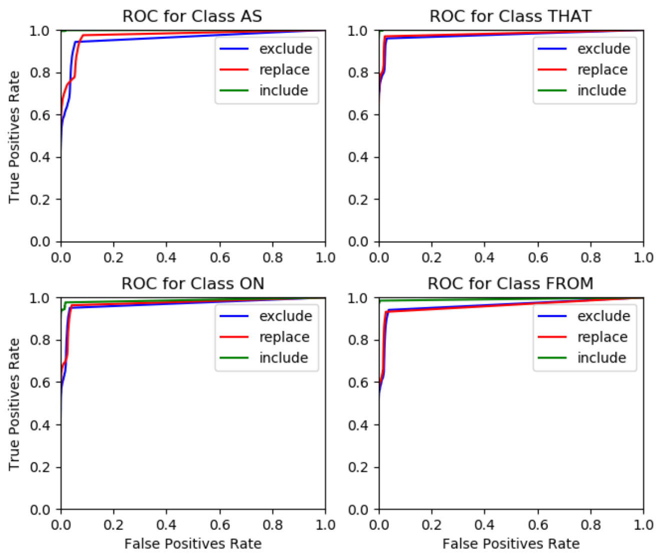

Figure 125 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the original preposition

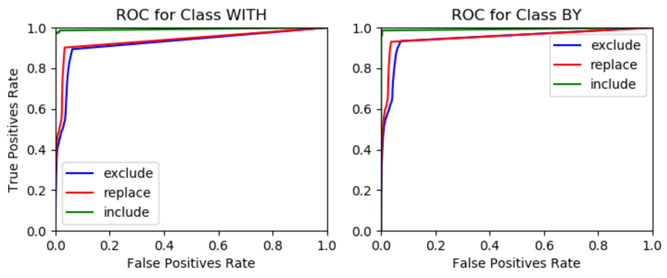

Figure 126 ROC for classes WITH (left) and BY (right) varying the original preposition

4.4.4.2.3 Metrics for Error Correction against a Threshold

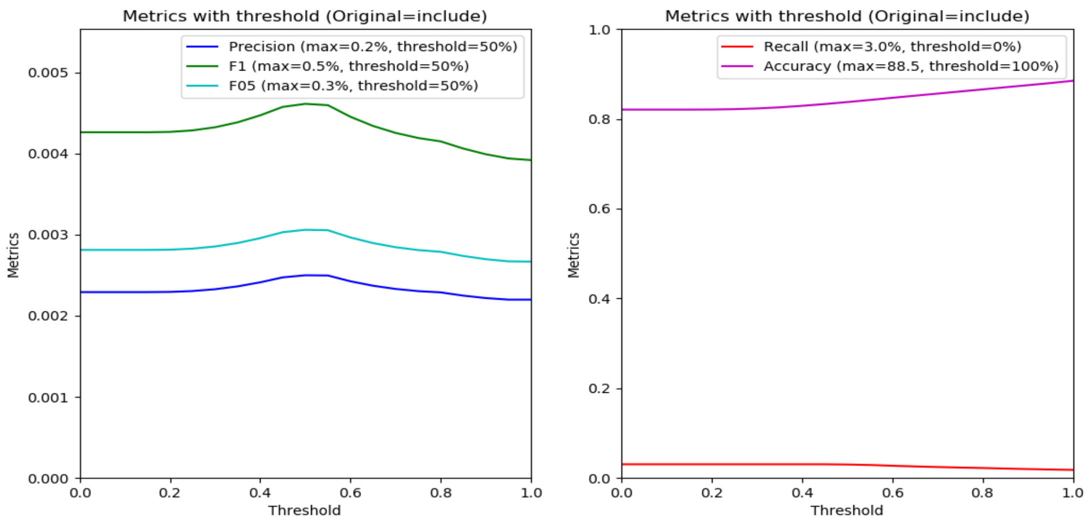

Figure 127 Correction Metrics against a Threshold including original preposition

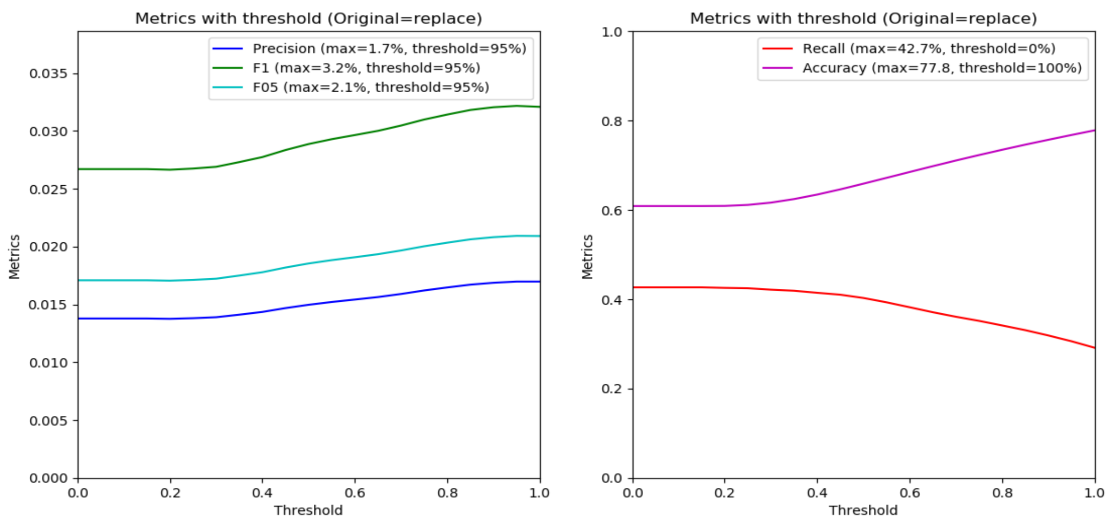

Figure 128 Correction Metrics against a Threshold replacing original preposition

4.4.4.2.4 Metrics for Error Detection and Correction Tasks

Original	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
Include	0.23%	3.05%	0.43%	0.28%	82.05%
Replace	1.37%	42.68%	2.67%	1.70%	60.89%
Exclude	1.10%	37.8%	2.14%	1.37%	56.64%

Table 38 Error correction for all classes varying the original preposition

Original	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
Include	1.14%	15.24%	2.13%	1.40%	82.33%
Replace	2.53%	78.65%	4.92%	3.15%	61.62%
Exclude	2.4%	82.32%	4.66%	2.98%	57.53%

Table 39 Error detection for all classes varying the original preposition

Original	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
Include	3.20%	3.65%	3.41%	3.28%	97.41%
Replace	2.30%	51.09%	4.40%	2.84%	72.33%
Exclude	1.73%	45.25%	3.33%	2.14%	67.27%

Table 40 Error correction for 10 top classes varying the original preposition

Original	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
Include	12.82%	14.60%	13.65%	13.14%	97.68%
Replace	3.71%	82.48%	7.10%	4.59%	73.00%
Exclude	3.34%	87.59%	6.44%	4.13%	68.15%

Table 41 Error detection for 10 top classes varying the original preposition

4.5 Embedding

4.5.1 One-Hot vs. Embedding

The comparison between the one-hot input vector and the embedding vector was carried out by setting the input type parameter at one-hot, while all other parameters kept their default values. The embedding experiment was already carried out in the hidden cells section.

4.5.1.1 Training Phase

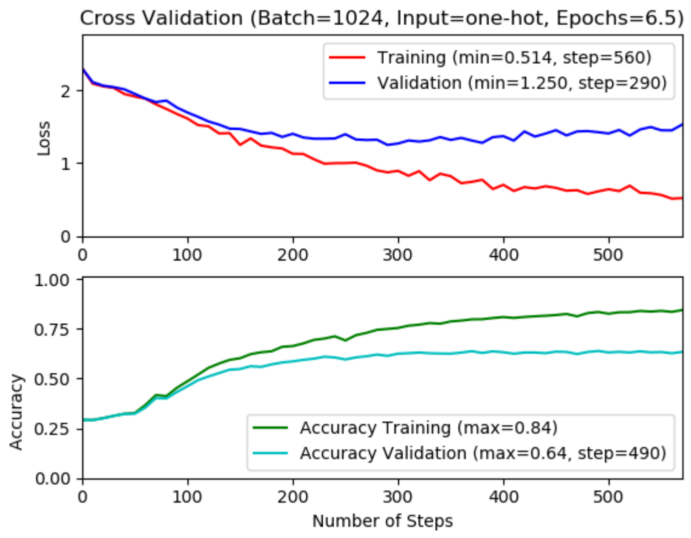

Figure 129 Cross validation using one-hot vectors as inputs

4.5.1.2 Testing Phase

4.5.1.2.1 Precision-Recall Graphs

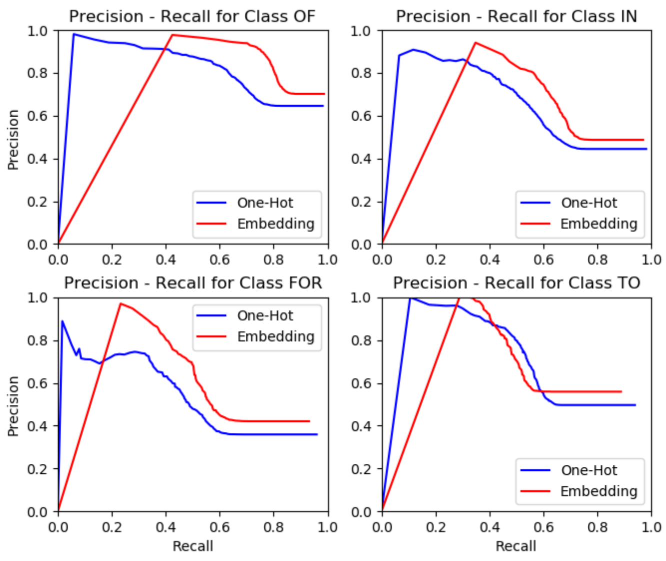

Figure 130 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) for one-hot vs. embedding input vectors

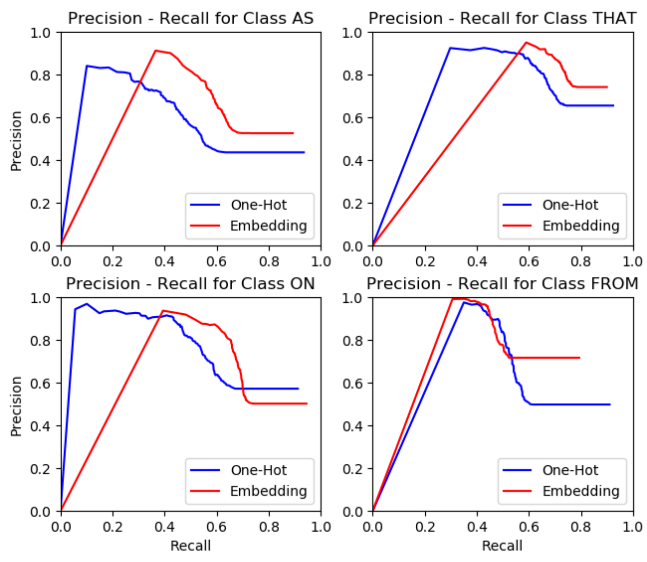

Figure 131 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) for one-hot vs. embedding input vectors

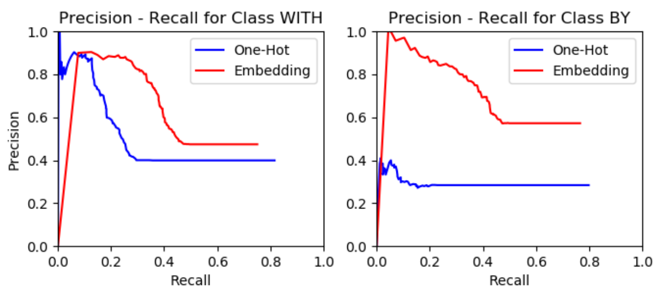

Figure 132 Precision vs. Recall for classes WITH (left) and BY (right) for one-hot vs. embedding input vectors

4.5.1.2.2 ROC Graphs

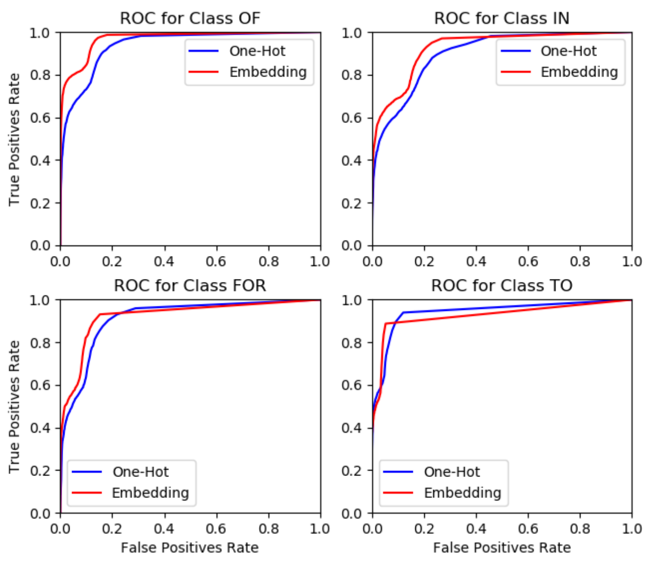

Figure 133 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) for one-hot vs. embedding input vectors

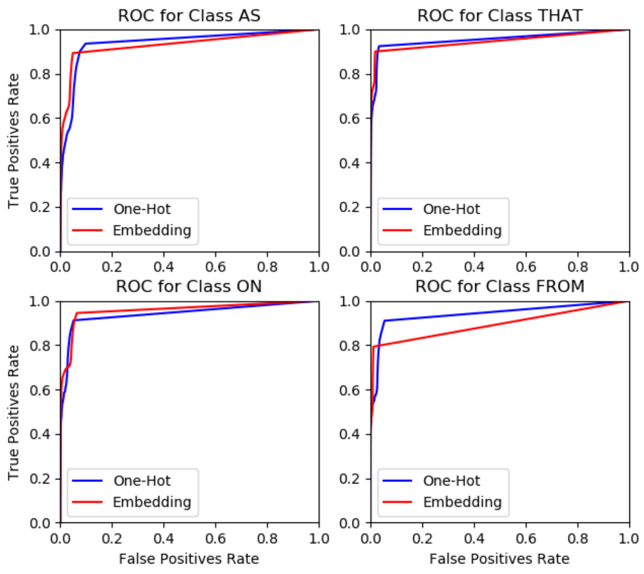

Figure 134 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) for one-hot vs. embedding input vectors

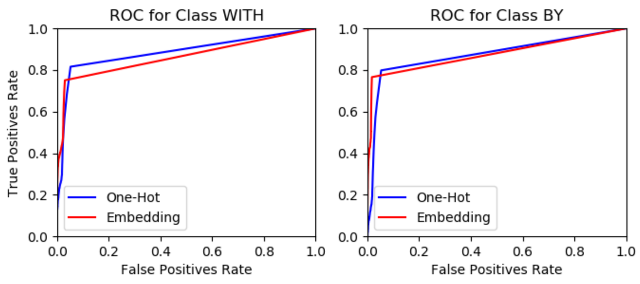

Figure 135 ROC for classes WITH (left) and BY (right) for one-hot vs. embedding input vectors

4.5.1.2.3 Metrics for Error Correction against a Threshold

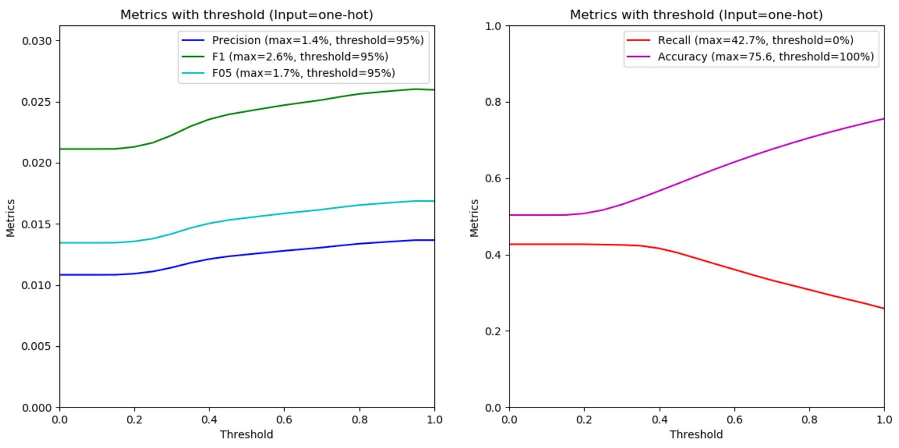

Figure 136 Correction Metrics against a Threshold using one-hot vectors as inputs

4.5.1.2.4 Metrics for Error Detection and Correction

Input	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
One-Hot	1.08%	4.68%	2.11%	1.34%	50.34%
Embedding	1.10%	37.8%	2.14%	1.37%	56.64%

Table 42 Error correction for all classes for one-hot vs. embedding input vectors

Input	Precision	Recall	F1	F05	Accuracy
One-Hot	2.21%	87.19%	4.31%	2.75%	51.18%
Embedding	2.4%	82.32%	4.66%	2.98%	57.53%

Table 43 Error detection for all classes for one-hot vs. embedding input vectors

Input	Precision	Recall	F1	F05	Accuracy
One-Hot	1.58%	51.09%	3.07%	1.96%	59.80%
Embedding	1.73%	45.25%	3.33%	2.14%	67.27%

Table 44 Error correction for 10 top classes for one-hot vs. embedding input vectors

Input	Precision	Recall	F1	F05	Accuracy
One-Hot	2.76%	89.05%	5.35%	3.42%	60.56%
Embedding	3.34%	87.59%	6.44%	4.13%	68.15%

Table 45 Error detection for 10 top classes for one-hot vs. embedding input vectors

4.5.2 Lexical Embedding Sizes

The size of the embedding input was varied at values of 50 and 200 per experiment, while all other parameters kept their default values. The experiment for a lexical embedding size of 100 was already carried out in the hidden cells section.

4.5.2.1 Training Phase

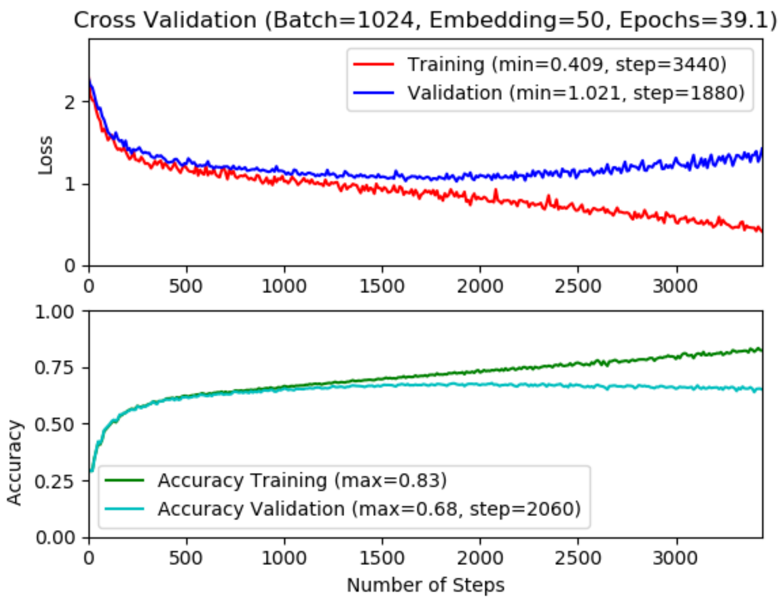

Figure 137 Cross validation for a lexical embedding size of 50

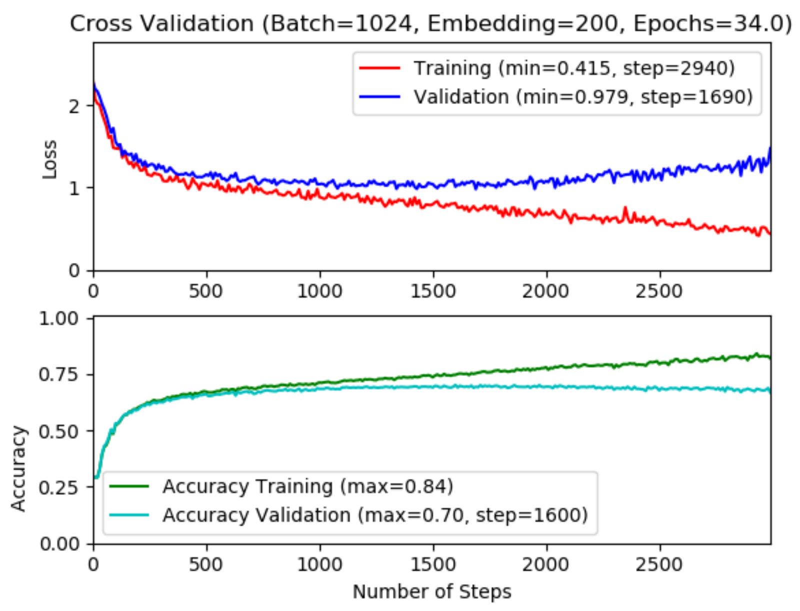

Figure 138 Cross validation for a lexical embedding size of 200

4.5.2.2 Testing Phase

4.5.2.2.1 Precision-Recall Graphs

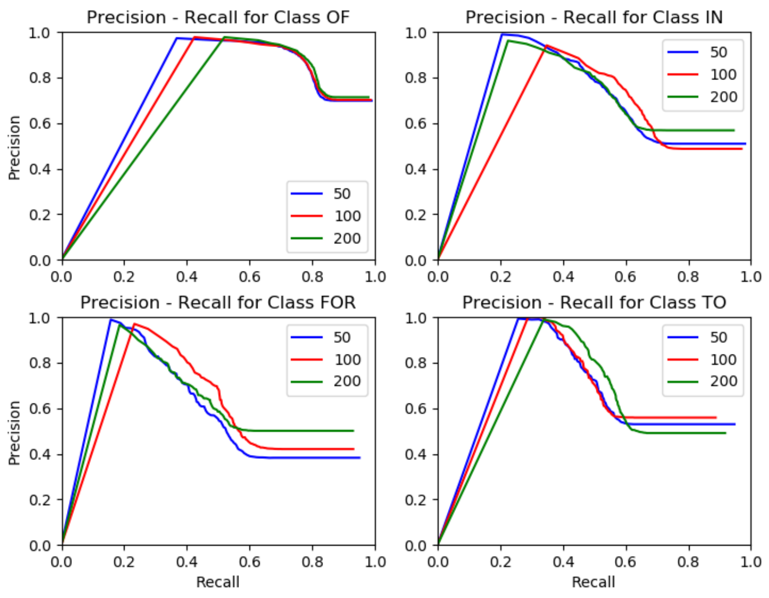

Figure 139 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the lexical embedding size of the input feature

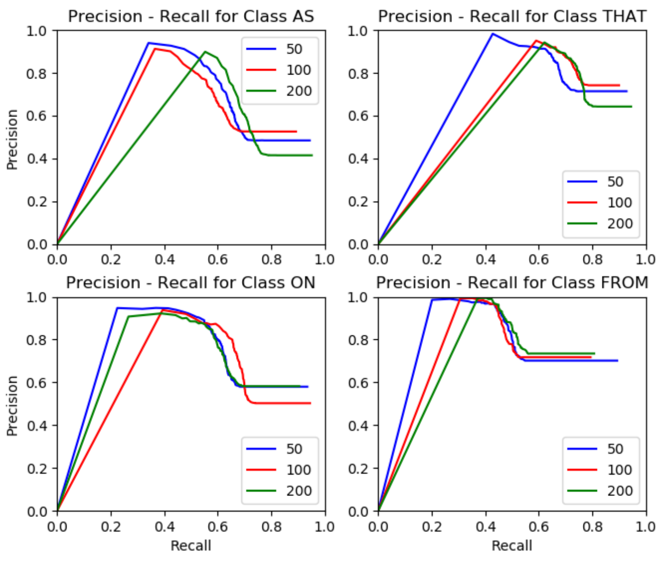

Figure 140 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the lexical embedding size of the input feature

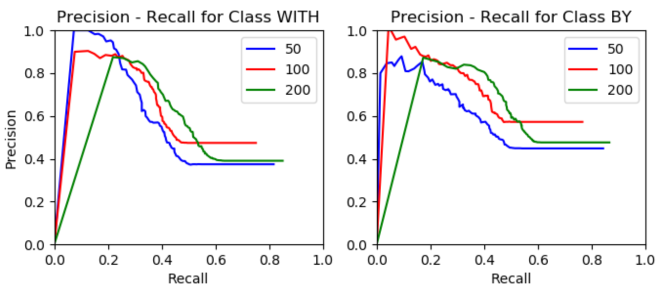

Figure 141 Precision vs. Recall for classes WITH (left) and BY (right) varying the lexical embedding size of the input feature

4.5.2.2.2 ROC Graphs

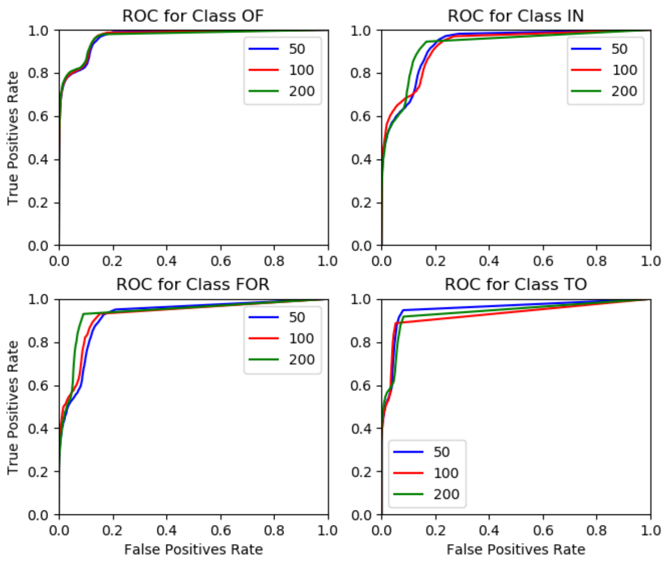

Figure 142 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the lexical embedding size of the input feature

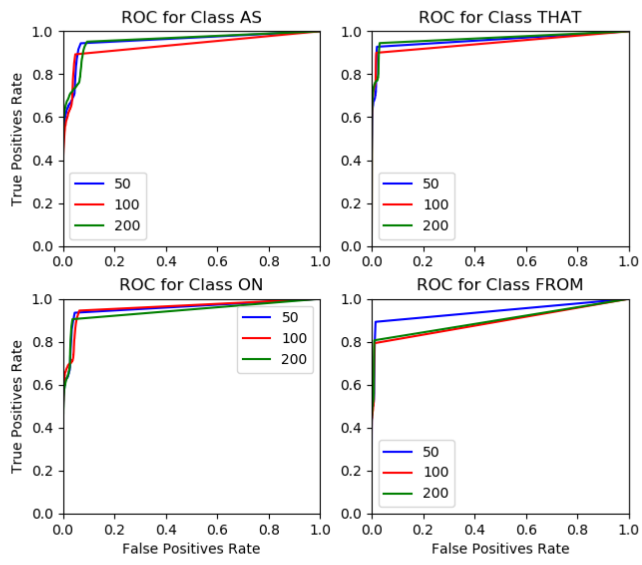

Figure 143 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the lexical embedding size of the input feature

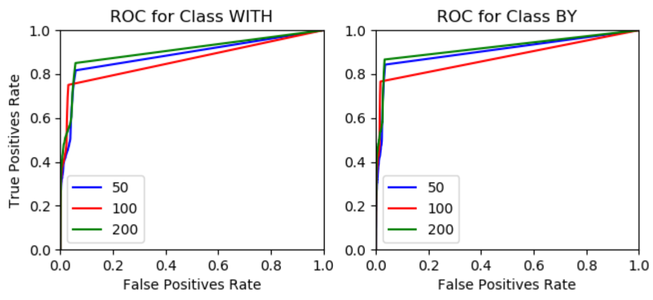

Figure 144 for classes WITH (left) and BY (right) varying the lexical embedding size of the input feature

4.5.2.2.3 Metrics for Error Correction against a Threshold

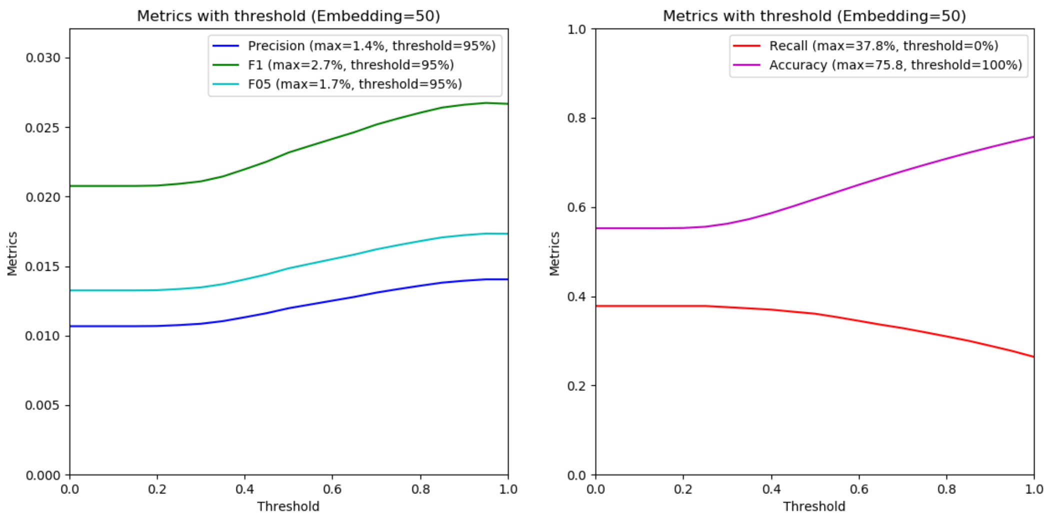

Figure 145 Correction Metrics against a Threshold for a lexical embedding size of 50

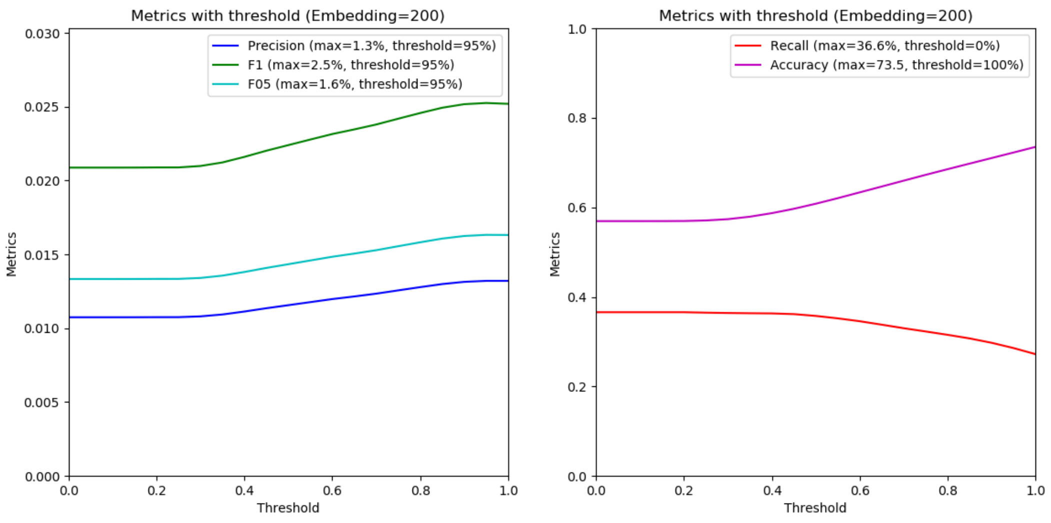

Figure 146 Correction Metrics against a Threshold for a lexical embedding size of 200

4.5.2.2.4 Metrics for Error Detection and Correction Tasks

Lexical Embedding Size	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
50	1.06%	37.80%	2.07%	1.32%	55.23%
100	1.10%	37.8%	2.14%	1.37%	56.64%
200	1.07%	36.58%	2.09%	1.33%	56.94%

Table 46 Error correction for all classes varying the lexical embedding size of the input feature

Lexical Embedding Size	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
50	2.29%	81.09%	4.45%	2.84%	56.08%
100	2.4%	82.32%	4.66%	2.98%	57.53%
200	2.43%	82.93%	4.73%	3.02%	57.82%

Table 47 Error detection for all classes varying the lexical embedding size of the input feature

Lexical Embedding Size	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
50	1.64%	45.25%	3.17%	2.04%	65.62%
100	1.73%	45.25%	3.33%	2.14%	67.27%
200	1.69%	43.79%	3.26%	2.09%	67.63%

Table 48 Error correction for 10 top classes varying the lexical embedding size of the input feature

Lexical Embedding Size	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
50	3.00%	82.48%	5.79%	3.71%	66.39%
100	3.34%	87.59%	6.44%	4.13%	68.15%
200	3.35%	86.86%	6.46%	4.15%	68.53%

Table 49 Error detection for 10 top classes varying the lexical embedding size of the input feature

4.5.3 Window Embedding Size

The window size for training the embedding size was varied at values of 1, 3 and 4 for each experiment, while all other parameters kept their default values. The embedding training for a window size of 2 was already carried out in the hidden cells section.

4.5.3.1 Training Phase

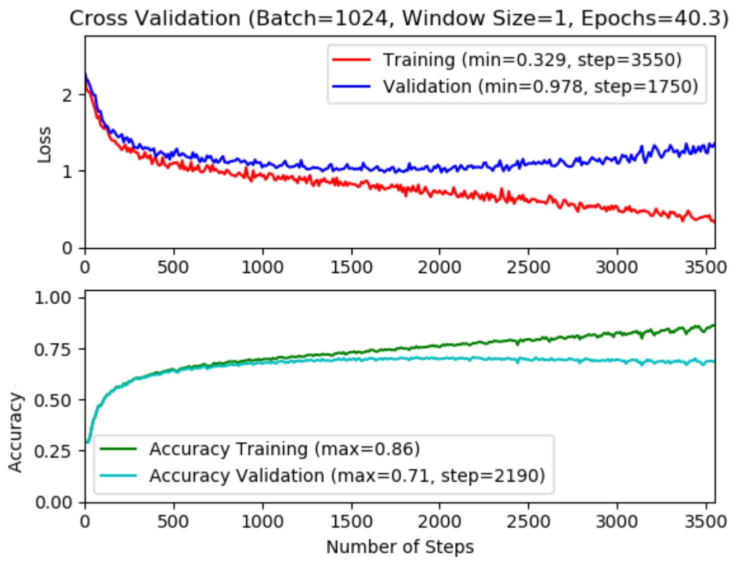

Figure 147 Cross validation embedding lexical features at a window size of 1

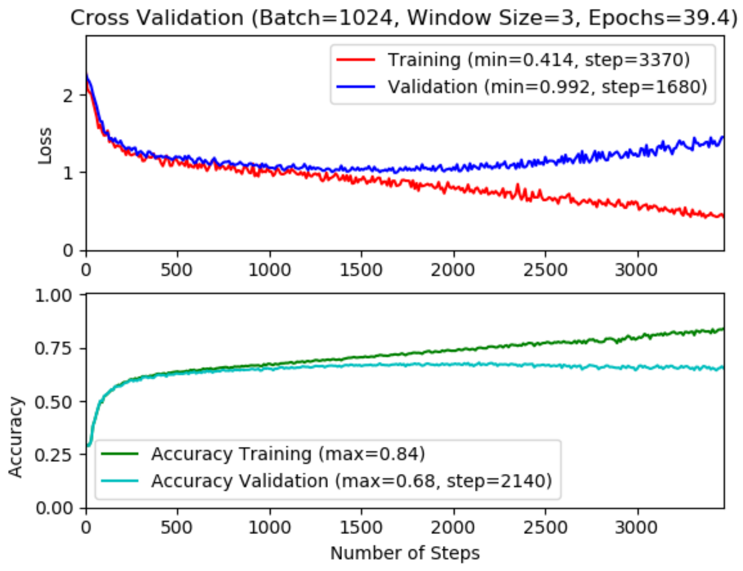

Figure 148 Cross validation embedding lexical features at a window size of 3

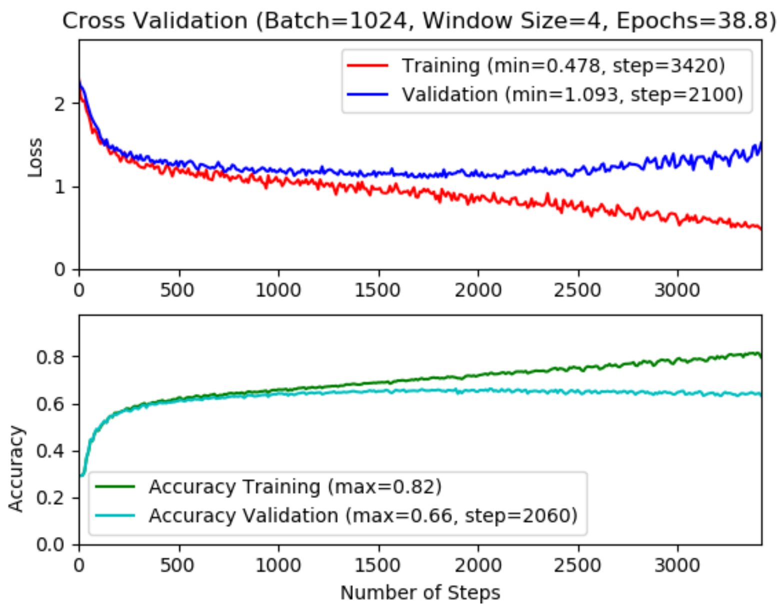

Figure 149 Cross validation embedding lexical features at a window size of 4

4.5.3.2 Testing Phase

4.5.3.2.1 Precision-Recall Graphs

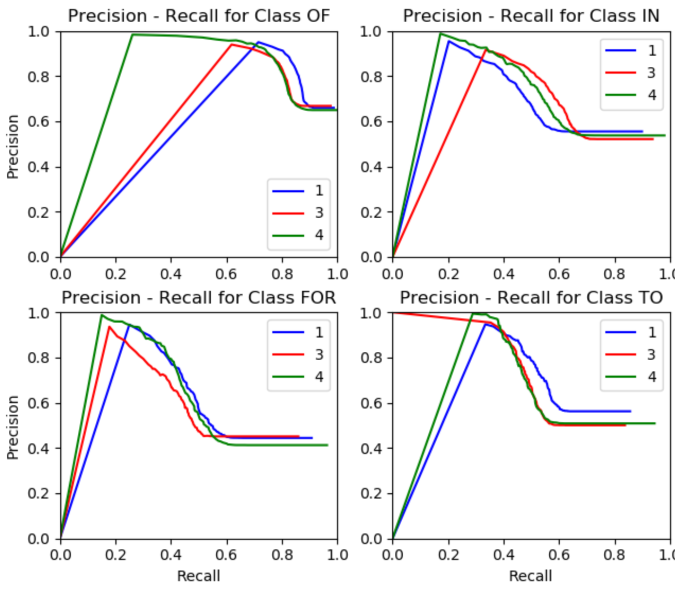

Figure 150 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) by embedding lexical features at different window sizes

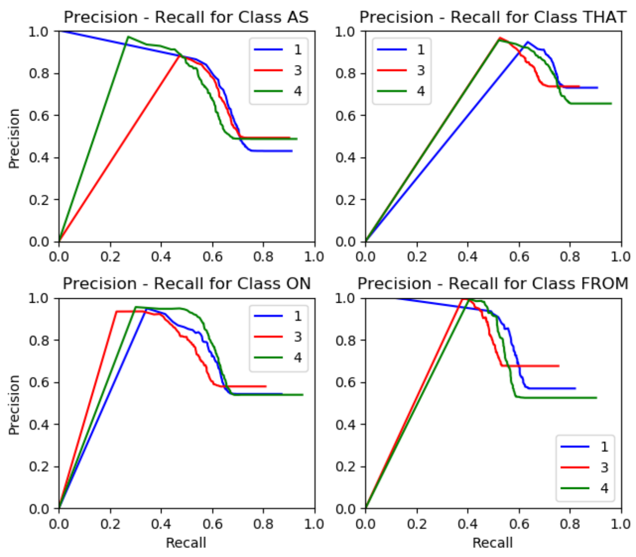

Figure 151 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) by embedding lexical features at different window sizes

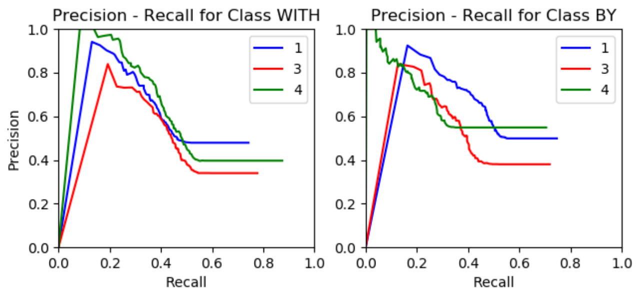

Figure 152 Precision vs. Recall for classes WITH (left) and BY (right) by embedding lexical features at different window sizes

4.5.3.2.2 ROC Graphs

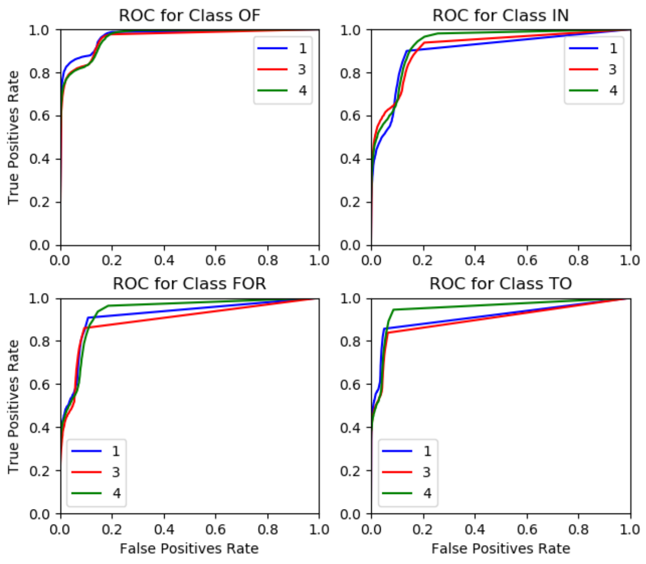

Figure 153 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) by embedding lexical features at different window sizes

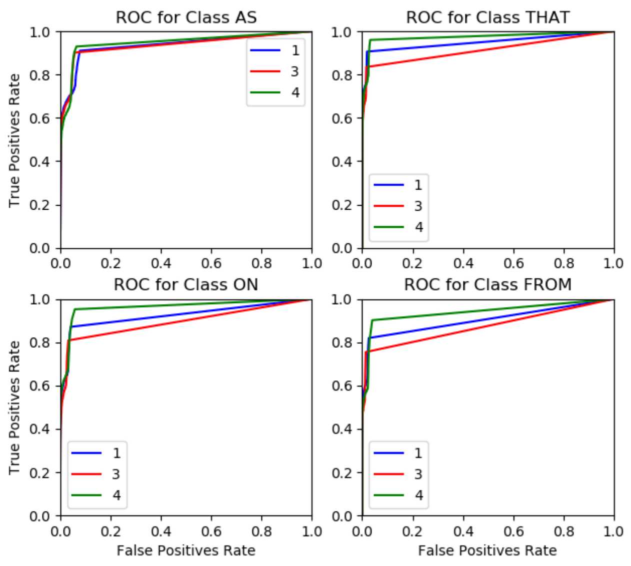

Figure 154 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) by embedding lexical features at different window sizes

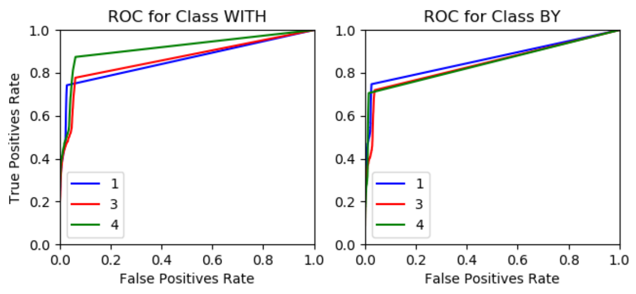

Figure 155 ROC for classes WITH (left) and BY (right) by embedding lexical features at different window sizes

4.5.3.2.3 Metrics for Error Correction against a Threshold

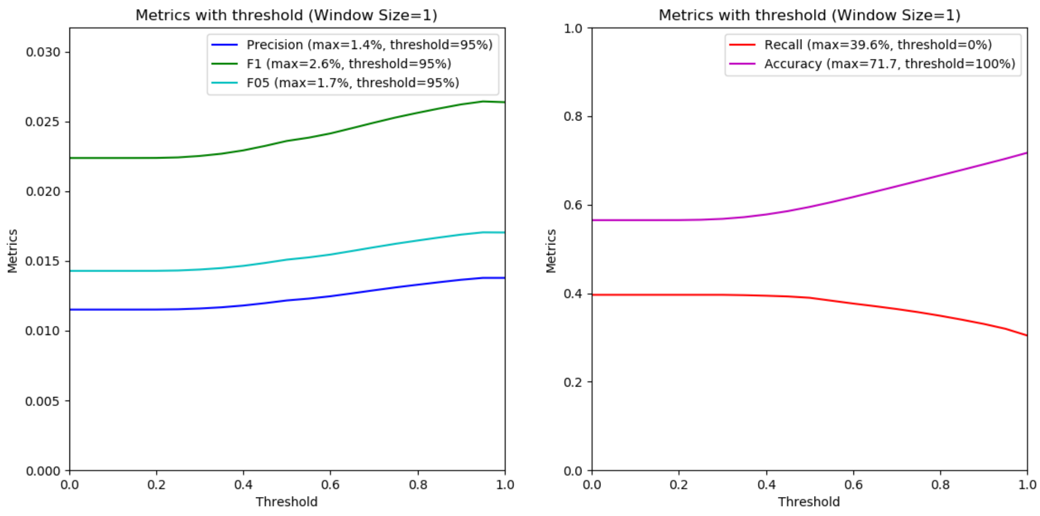

Figure 156 Correction Metrics against a Threshold embedding lexical features at a window size of 1

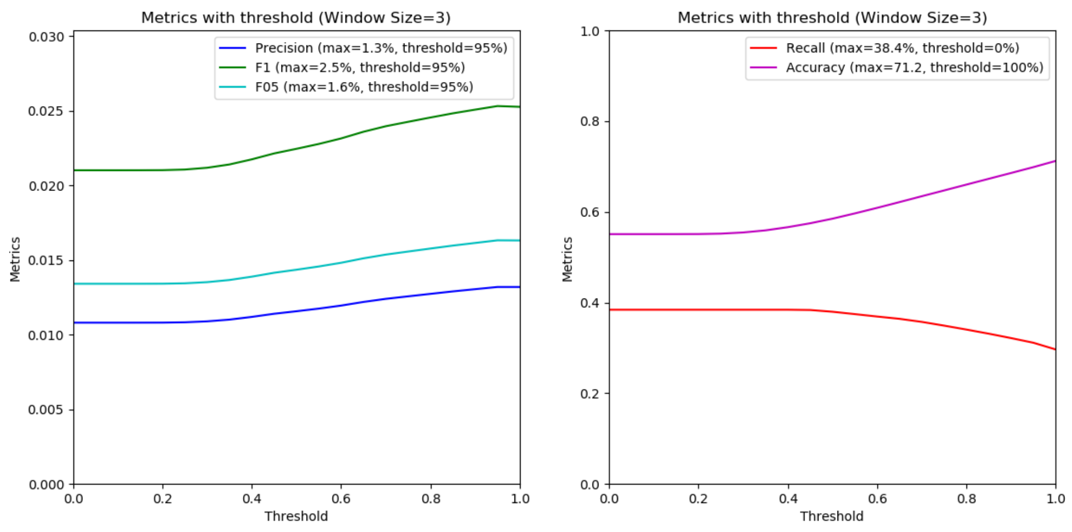

Figure 157 Correction Metrics against a Threshold embedding lexical features at a window size of 3

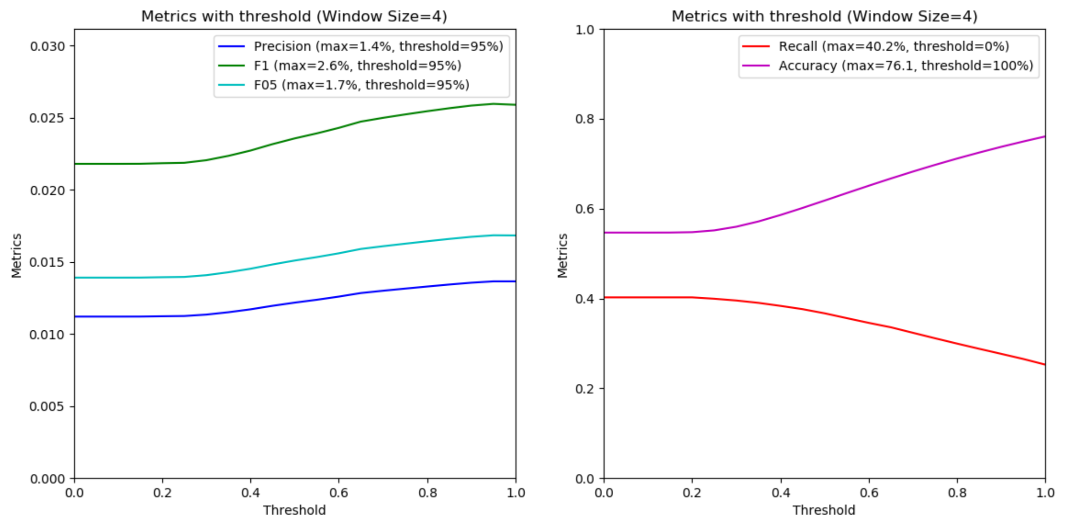

Figure 158 Correction Metrics against a Threshold embedding lexical features at a window size of 4

4.5.3.2.4 Metrics for Error Detection and Correction Tasks

Window	Precision	Recall	F1	F05	Accuracy
1	1.15%	39.63%	2.24%	1.43%	56.51%
2	1.10%	37.8%	2.14%	1.37%	56.64%
3	1.08%	38.41%	2.10%	1.34%	55.09%
4	1.12%	40.24%	2.18%	1.39%	54.67%

Table 50 Error correction for all classes by embedding lexical features at different window sizes

Sequence	Precision	Recall	F1	F05	Accuracy
1	2.35%	81.10%	4.58%	2.92%	57.33%
2	2.4%	82.32%	4.66%	2.98%	57.53%
3	2.33%	82.93%	5.54%	2.89%	55.96%
4	2.34%	84.14%	4.56%	2.91%	55.53%

Table 51 Error detection for all classes by embedding lexical features at different window sizes

Sequence	Precision	Recall	F1	F05	Accuracy
1	1.80%	47.44%	3.47%	2.23%	67.13%
2	1.73%	45.25%	3.33%	2.14%	67.27%

3	1.66%	45.98%	3.21%	2.06%	65.44%
4	1.71%	48.17%	3.31%	2.12%	64.96%

Table 52 Error correction for 10 top classes by embedding lexical features at different window sizes

Sequence	Precision	Recall	F1	F05	Accuracy
1	3.16%	83.21%	6.09%	3.91%	67.88%
2	3.34%	87.59%	6.44%	4.13%	68.15%
3	3.08%	85.40%	5.96%	3.82%	66.25%
4	3.01%	84.67%	5.82%	3.73%	65.72%

Table 53 Error detection for 10 top classes by embedding lexical features at different window sizes

4.6 Models

4.6.1 Simple vs. Bidirectional

The comparison between the two different LSTM models was carried out by switching the model type to bidirectional, while all other parameters kept their default values. The simple model experiment was already carried out in the hidden cells section.

4.6.1.1 Training Phase

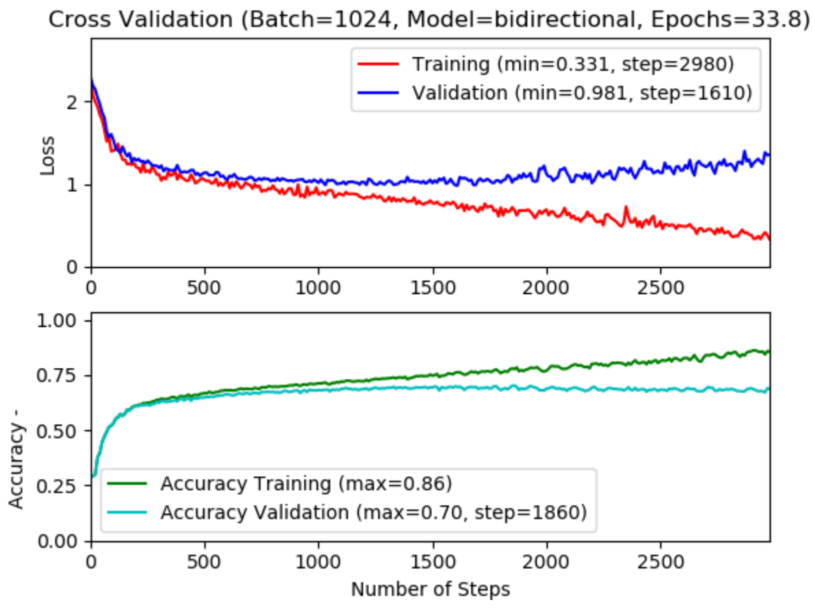

Figure 159 Cross validation using a bidirectional LSTM model

4.6.1.2 Testing Phase

4.6.1.2.1 Precision-Recall Graphs

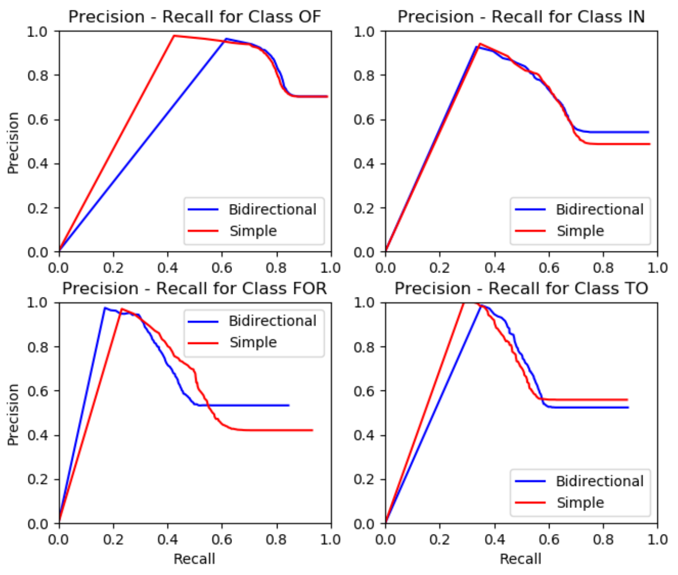

Figure 160 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the core LSTM model

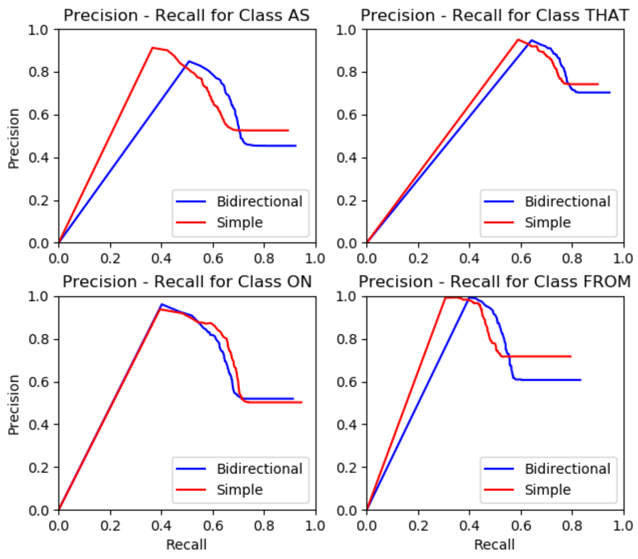

Figure 161 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the core LSTM model

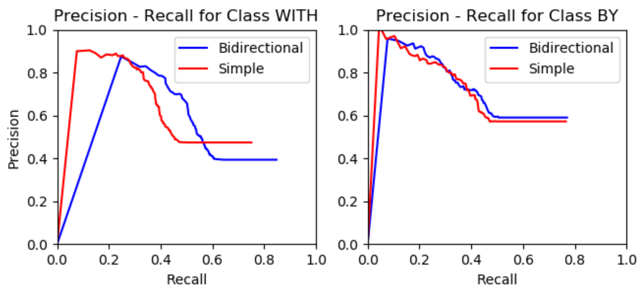

Figure 162 Precision vs. Recall for classes WITH (left) and BY (right) varying the core LSTM model

4.6.1.2.2 ROC Graphs

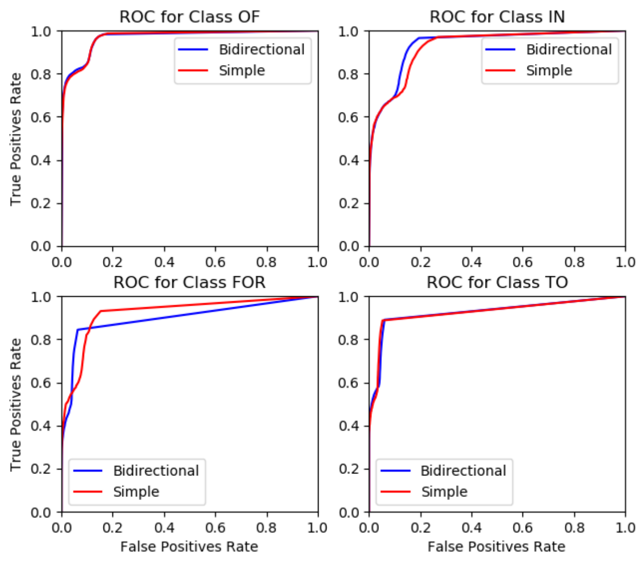

Figure 163 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the core LSTM model

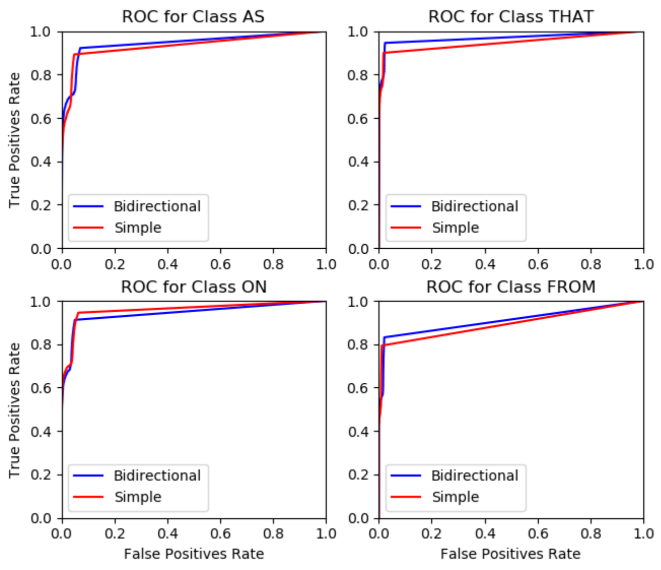

Figure 164 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the core LSTM model

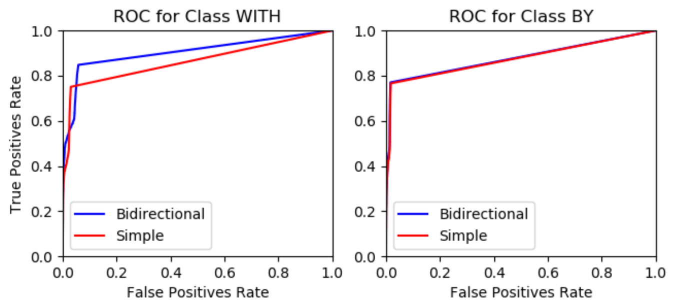

Figure 165 ROC for classes WITH (left) and BY (right) varying the core LSTM model

4.6.1.2.3 Metrics for Error Correction against a Threshold

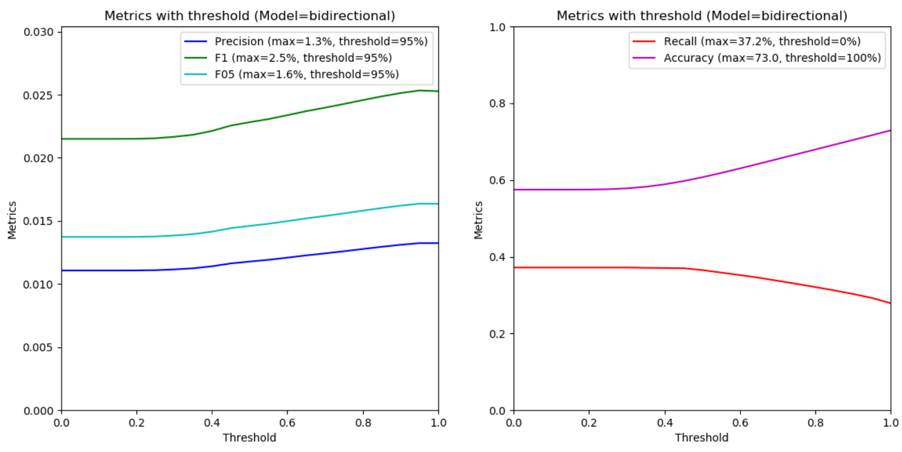

Figure 166 Correction Metrics against a Threshold using a bidirectional LSTM model

4.6.1.2.4 Metrics for Error Detection and Correction Tasks

Model	Precision	Recall	F1	F05	Accuracy
simple	1.10%	37.8%	2.14%	1.37%	56.64%
Bidirectional	1.11%	37.19%	2.15%	1.37%	57.50%

Table 54 Error correction for all classes varying the core LSTM model

Model	Precision	Recall	F1	F05	Accuracy
Simple	2.4%	82.32%	4.66%	2.98%	57.53%
Bidirectional	2.38%	79.88%	4.62%	2.95%	58.35%

Table 55 Error detection for all classes varying the core LSTM model

Model	Precision	Recall	F1	F05	Accuracy
Simple	1.73%	45.25%	3.33%	2.14%	67.27%
Bidirectional	1.76%	44.52%	3.38%	1.17%	68.31%

Table 56 Error correction for 10 top classes varying the core LSTM model

Model	Precision	Recall	F1	F05	Accuracy
Simple	3.34%	87.59%	6.44%	4.13%	68.15%
Bidirectional	3.22%	81.75%	6.21%	3.99%	69.09%

Table 57 Error detection for 10 top classes varying the core LSTM model

4.6.2 Attention

The attention mechanism was tested by switching the attention parameter to True and setting the maximum sequence length at 20, while all other parameters kept their default values. The experiment without using the attention mechanism and with a maximum sequence length of 20 was already carried out in the length of the input sequence section.

4.6.2.1 *Training Phase*

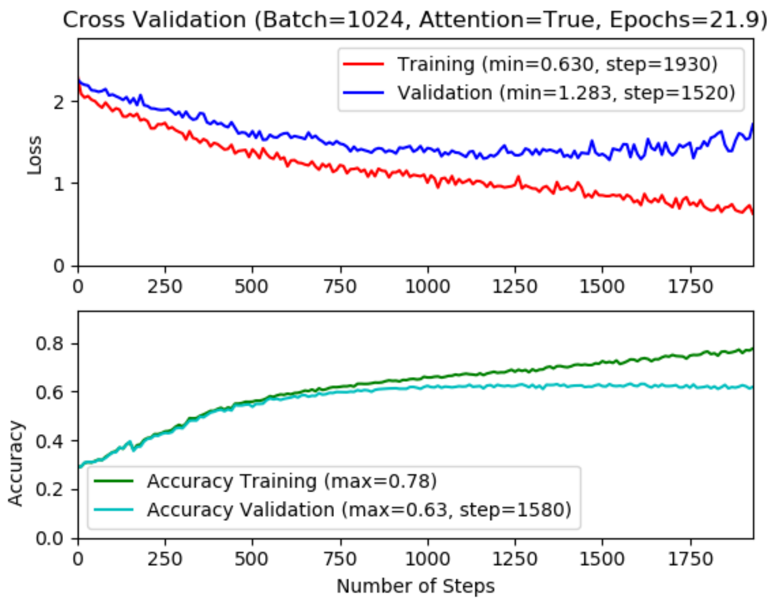

Figure 167 Cross validation using attention mechanism

4.6.2.2 *Testing Phase*

4.6.2.2.1 Precision-Recall Graphs

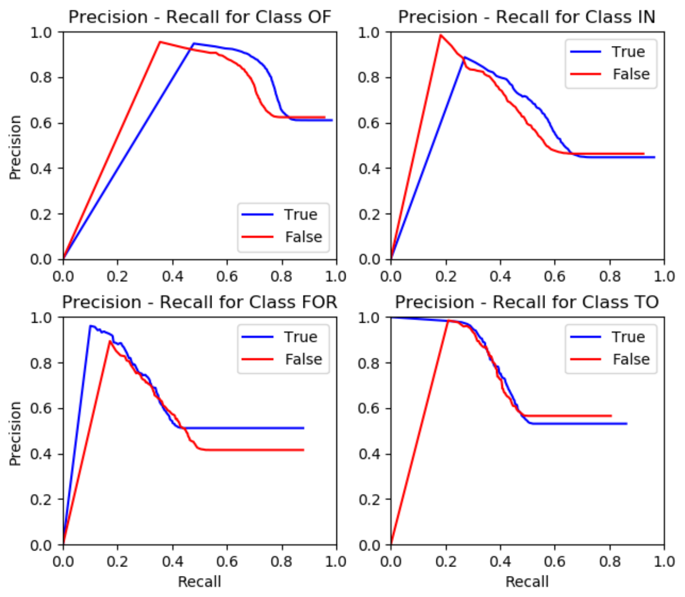

Figure 168 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) comparing the use of attention mechanism

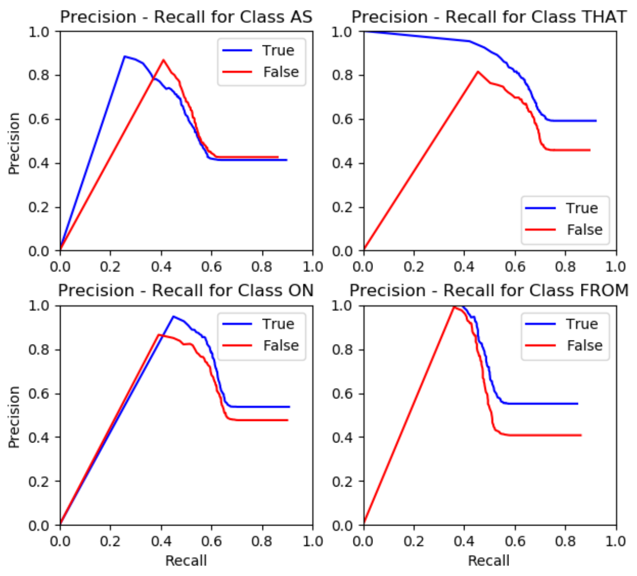

Figure 169 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) comparing the use of attention mechanism

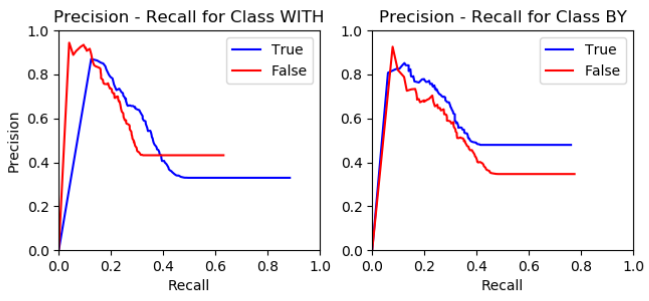

Figure 170 Precision vs. Recall for classes WITH (left) and BY (right) comparing the use of attention mechanism

4.6.2.2.2 ROC Graphs

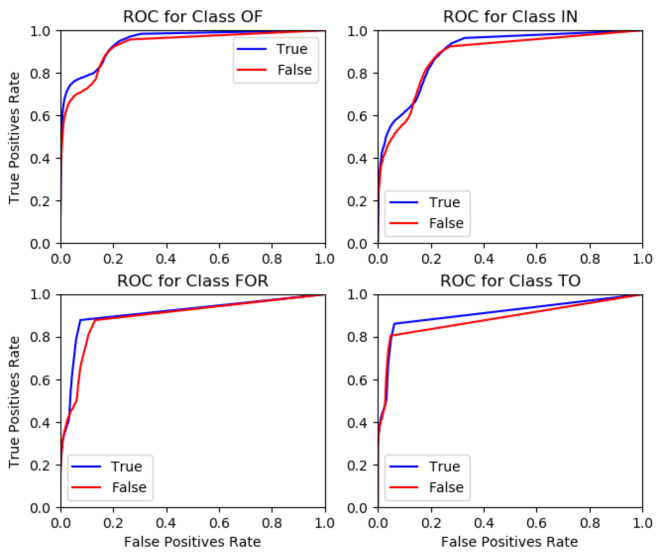

Figure 171 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) comparing the use of attention mechanism

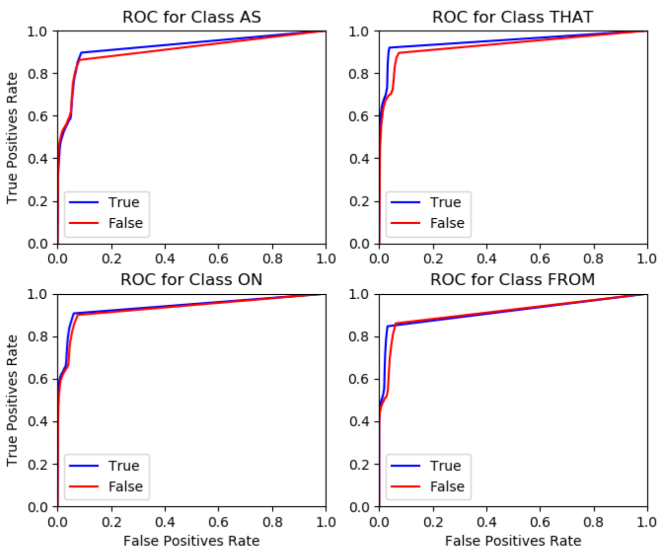

Figure 172 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) comparing the use of attention mechanism

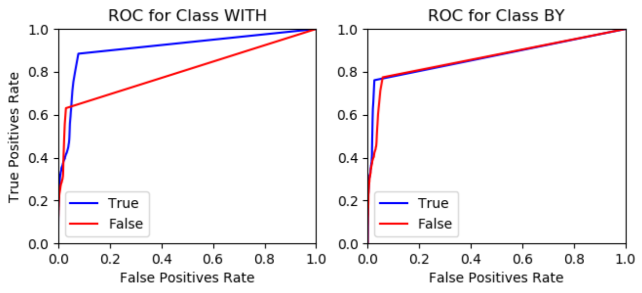

Figure 173 ROC for classes WITH (left) and BY (right) comparing the use of attention mechanism

4.6.2.2.3 Metrics for Error Correction against a Threshold

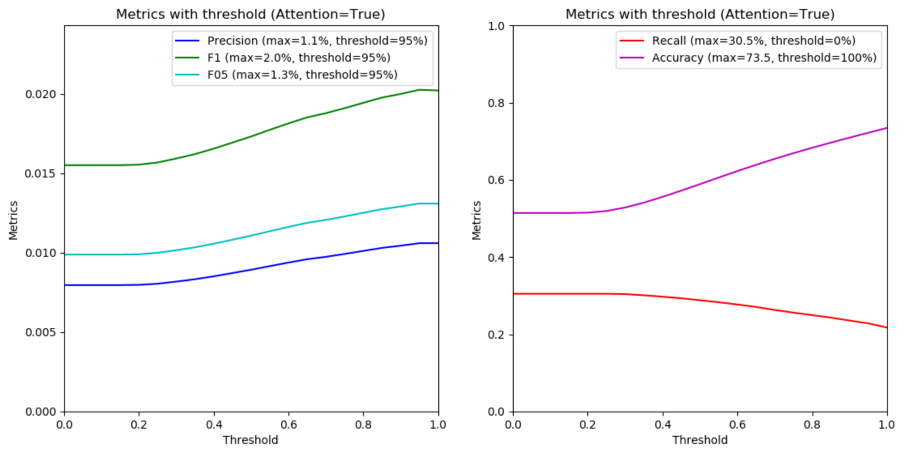

Figure 174 Correction Metrics against a Threshold using attention mechanism

4.6.2.2.4 Metrics for Error Detection and Correction Tasks

Attention	Precision	Recall	F1	F05	Accuracy
True	0.79%	30.49%	1.55%	0.99%	51.41%
False	0.70%	28.05%	1.37%	0.87%	49.24%

Table 58 Error correction for all classes comparing the use of attention mechanism

Spelling	Precision	Recall	F1	F05	Accuracy
True	1.99%	76.22%	3.87%	2.47%	52.28%
False	1.94%	79.27%	3.86%	2.46%	50.20%

Table 59 Error detection for all classes comparing the use of attention mechanism

Spelling	Precision	Recall	F1	F05	Accuracy
True	1.17%	36.49%	2.28%	1.46%	61.07%
False	1.02%	33.58%	1.97%	1.26%	58.48%

Table 60 Error correction for 10 top classes comparing the use of attention mechanism

Spelling	Precision	Recall	F1	F05	Accuracy
True	2.47%	76.64%	4.79%	3.07%	61.88%
False	2.45%	81.02%	4.76%	3.04%	59.42%

Table 61 Error detection for 10 top classes comparing the use of attention mechanism

4.6.3 Dropout

Regularization through dropout was tested by switching the dropout parameter to True while varying the value of the keep probability parameter at values of 0.5, 0.6, 0.7, 0.8, 0.9 and 1.0 for each experiment. The number of hidden layers was also set at 1600 and the number of layers at 8, while all other parameters kept their default values.

4.6.3.1 Training Phase

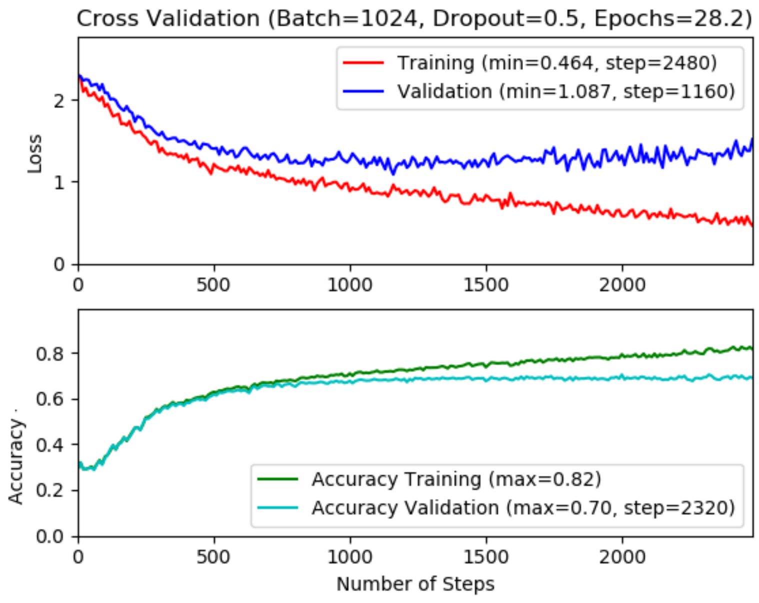

Figure 175 Cross validation keeping weights with a probability of 0.5

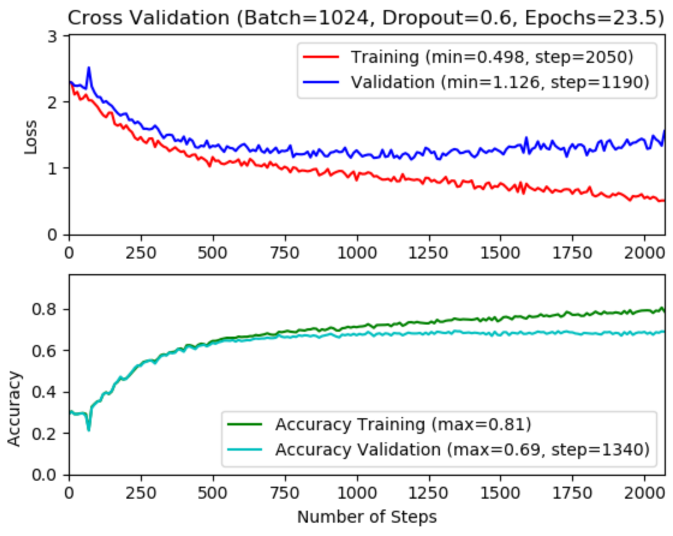

Figure 176 Cross validation keeping weights with a probability of 0.6

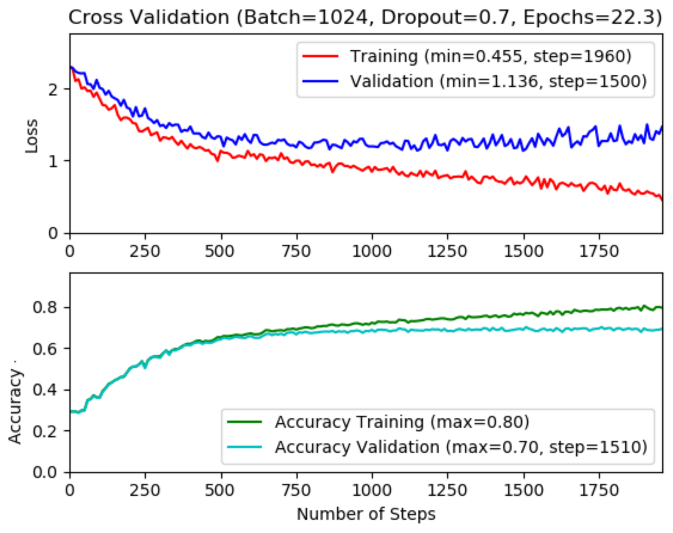

Figure 177 Cross validation keeping weights with a probability of 0.7

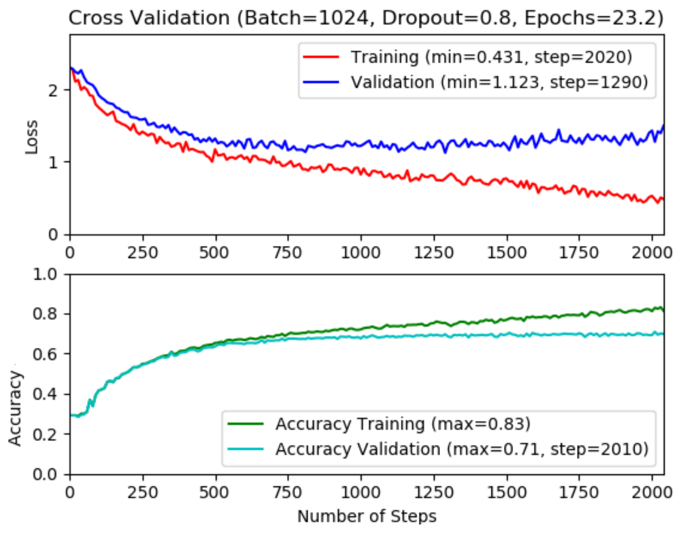

Figure 178 Cross validation keeping weights with a probability of 0.8

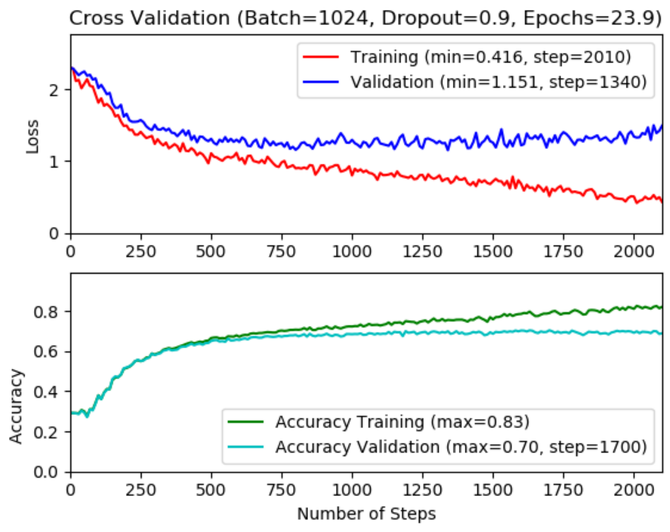

Figure 179 Cross validation keeping weights with a probability of 0.9

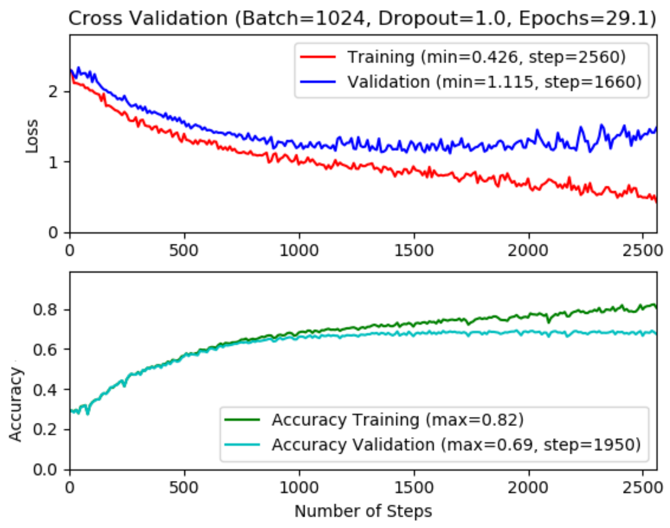

Figure 180 Cross validation keeping weights with a probability of 1.0

4.6.3.1.1 Precision-Recall Graphs

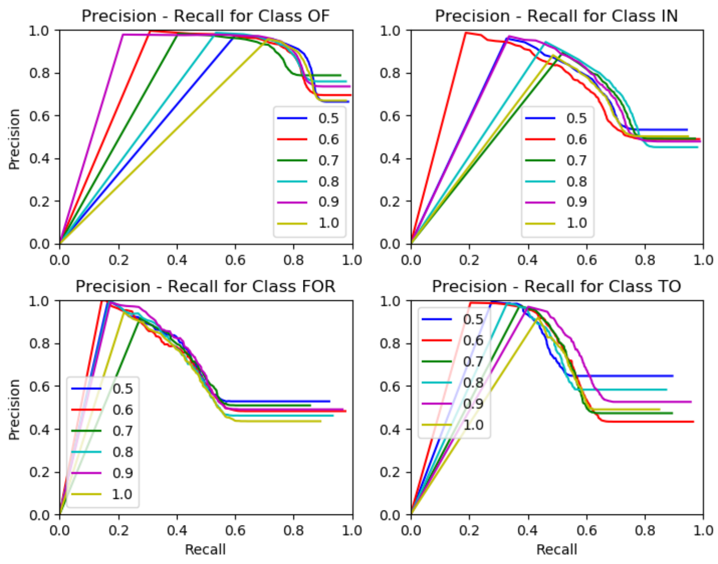

Figure 181 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the probability of keeping weights through dropout

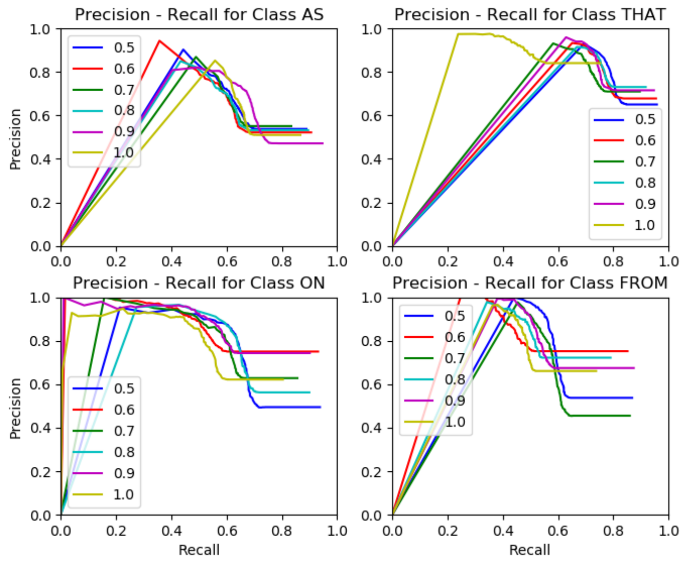

Figure 182 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the probability of keeping weights through dropout

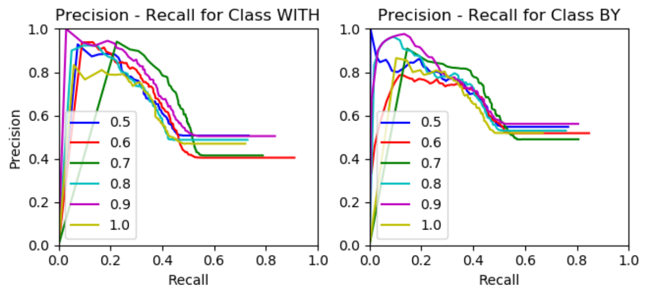

Figure 183 Precision vs. Recall for classes WITH (left) and BY (right) varying the probability of keeping weights through dropout

4.6.3.1.2 ROC Graphs

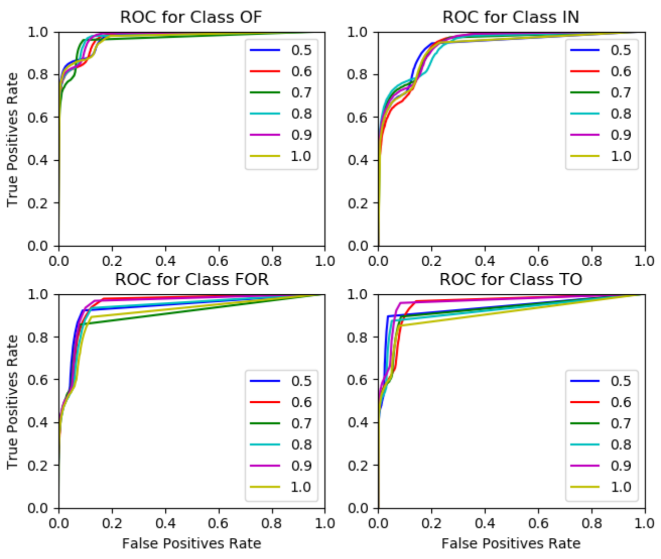

Figure 184 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the probability of keeping weights through dropout

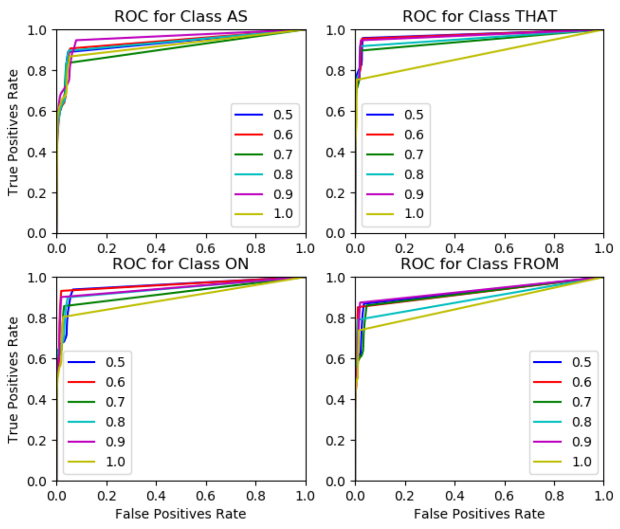

Figure 185 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the probability of keeping weights through dropout

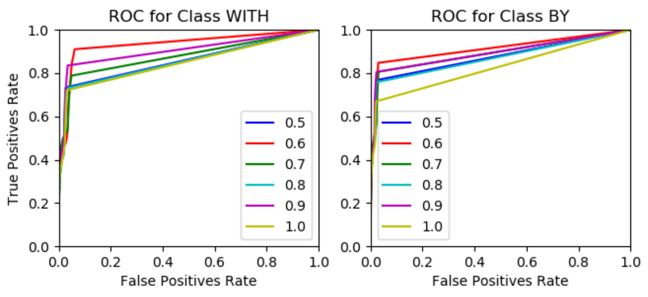

Figure 186 ROC for classes WITH (left) and BY (right) varying the probability of keeping weights through dropout

4.6.3.1.3 Metrics for Error Correction against a Threshold

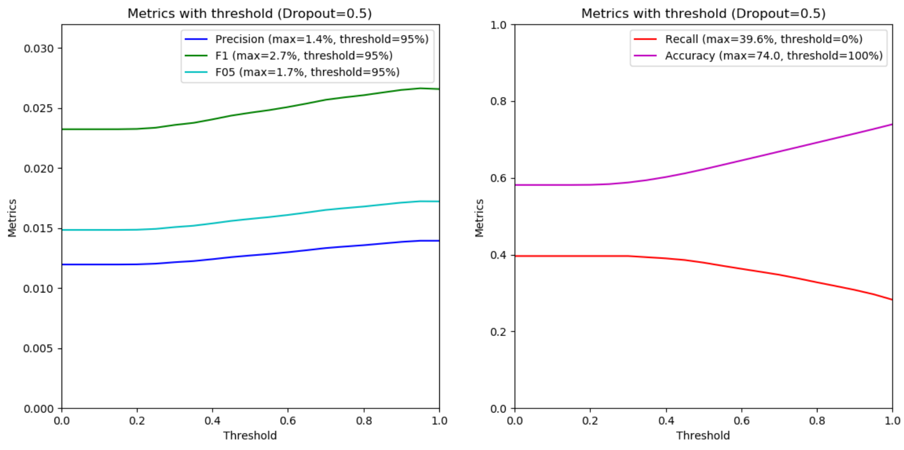

Figure 187 Correction Metrics against a Threshold keeping weights with a probability of 0.5

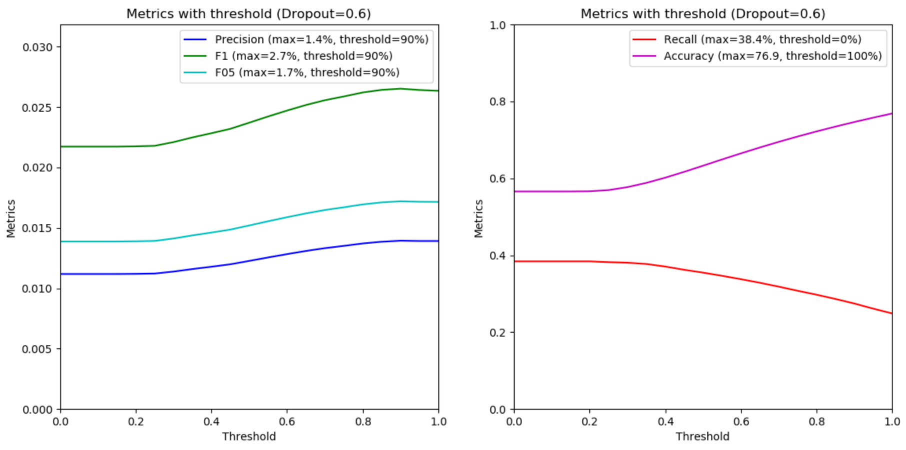

Figure 188 Correction Metrics against a Threshold keeping weights with a probability of 0.6

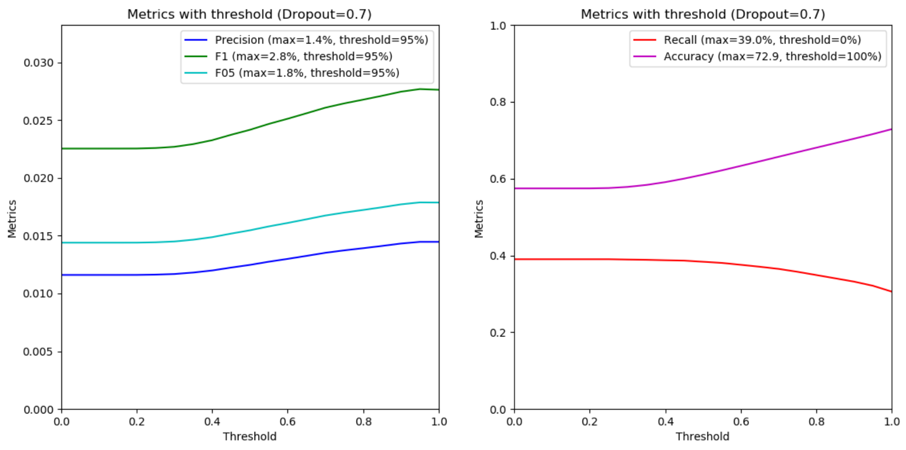

Figure 189 Correction Metrics against a Threshold keeping weights with a probability of 0.7

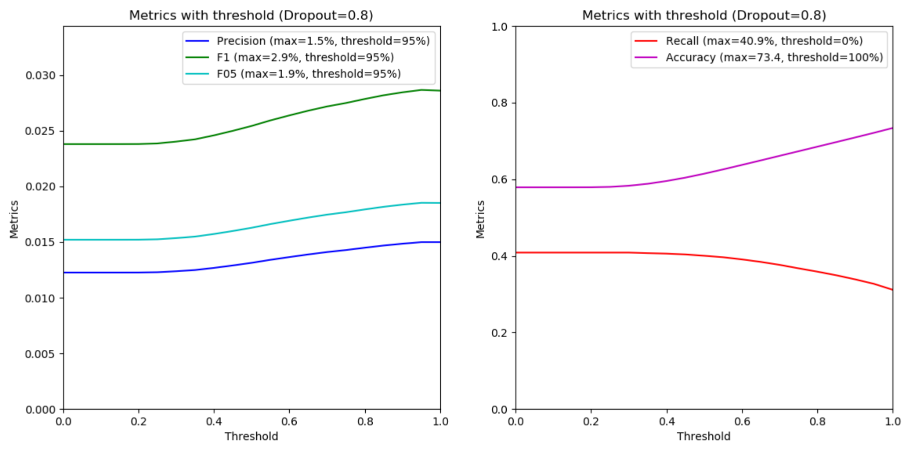

Figure 190 Correction Metrics against a Threshold keeping weights with a probability of 0.8

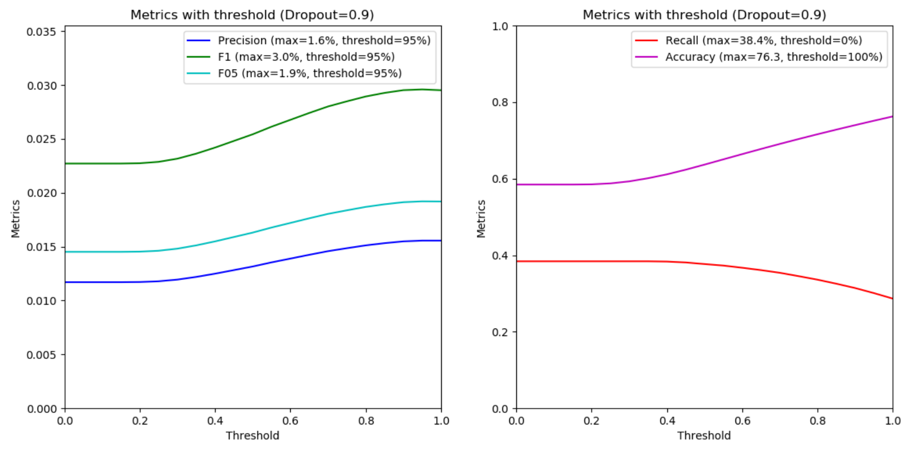

Figure 191 Correction Metrics against a Threshold keeping weights with a probability of 0.9

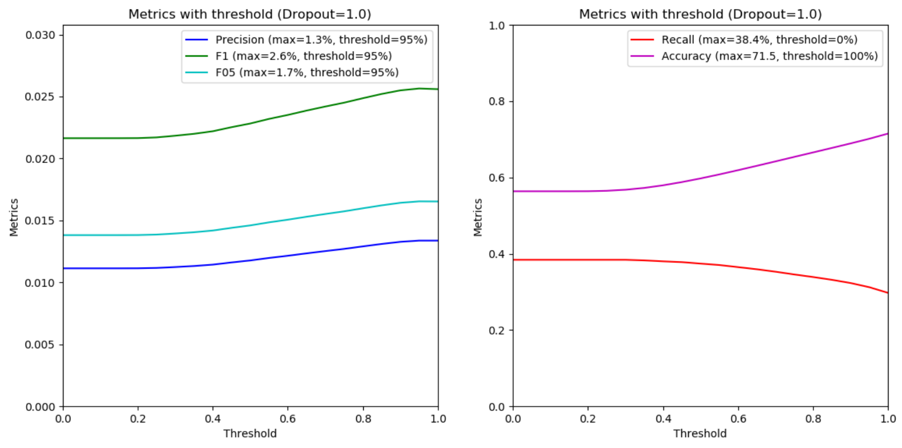

Figure 192 Correction Metrics against a Threshold keeping weights with a probability of 1.0

4.6.3.1.4 Metrics for Error Detection and Correction Tasks

Keep Probabilities	Precision	Recall	F1	F05	Accuracy
0.5	1.19%	39.63%	2.32%	1.48%	58.14%
0.6	1.12%	38.41%	2.17%	1.39%	56.59%
0.7	1.16%	39.02%	2.25%	1.44%	57.47%
0.8	1.22%	40.85%	2.38%	1.52%	57.90%

0.9	1.11%	36.58%	2.15%	1.38%	58.28%
1.0	1.11%	38.41%	2.16%	1.38%	56.38%

Table 62 Error correction for all classes varying the probability of keeping weights through dropout

Keep Probabilities	Precision	Recall	F1	F05	Accuracy
0.5	2.44%	81.10%	4.75%	3.04%	58.98%
0.6	2.38%	81.70%	4.62%	2.95%	57.44%
0.7	2.26%	76.22%	4.40%	2.81%	58.20%
0.8	2.34%	78.04%	4.54%	2.90%	58.64%
0.9	2.36%	78.05%	4.60%	2.94%	59.11%
1.0	2.30%	79.26%	4.46%	2.85%	57.19%

Table 63 Error detection for all classes varying the probability of keeping weights through dropout

Keep Probabilities	Precision	Recall	F1	F05	Accuracy
0.5	1.91%	47.44%	3.68%	2.27%	69.07%
0.6	1.75%	45.98%	3.38%	2.17%	67.22%
0.7	1.84%	46.71%	3.53%	2.27%	68.24%
0.8	1.95%	48.90%	3.76%	2.41%	68.77%
0.9	1.78%	43.79%	3.42%	2.20%	69.24%
1.0	1.74%	45.98%	3.35%	2.15%	66.97%

Table 64 Error correction for 10 top classes varying the probability of keeping weights through dropout

Keep Probabilities	Precision	Recall	F1	F05	Accuracy
0.5	3.38%	83.94%	6.51%	4.19%	65.84%
0.6	3.20%	83.94%	6.16%	3.96%	68.02%
0.7	3.21%	81.75%	6.18%	3.98%	68.98%
0.8	3.29%	82.48%	6.33%	4.08%	69.48%
0.9	3.27%	80.29%	6.28%	4.04%	70.01%
1.0	3.12%	82.48%	6.01%	3.86%	67.73%

Table 65 Error detection for 10 top classes varying the probability of keeping weights through dropout

4.6.4 Classes

The number of classes was tested by varying the parameter controlling the number of classes at values of 5, 15 and 20 per experiment, while keeping all other parameters at their default value. The experiment with a number of classes of 10 was already tested in the number of hidden cells section.

4.6.4.1 Training Phase

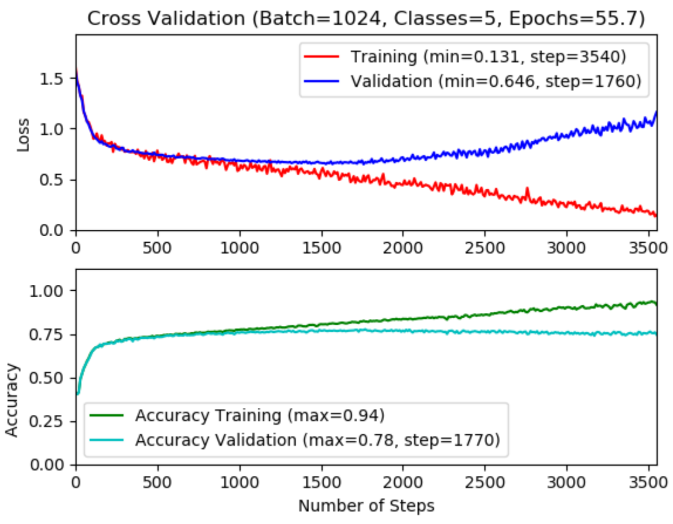

Figure 193 Cross validation aiming at 5 prepositional classes

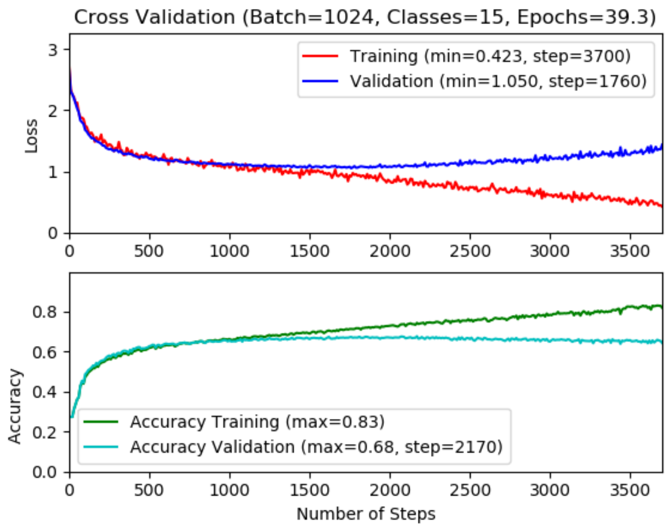

Figure 194 Cross validation aiming at 15 prepositional classes

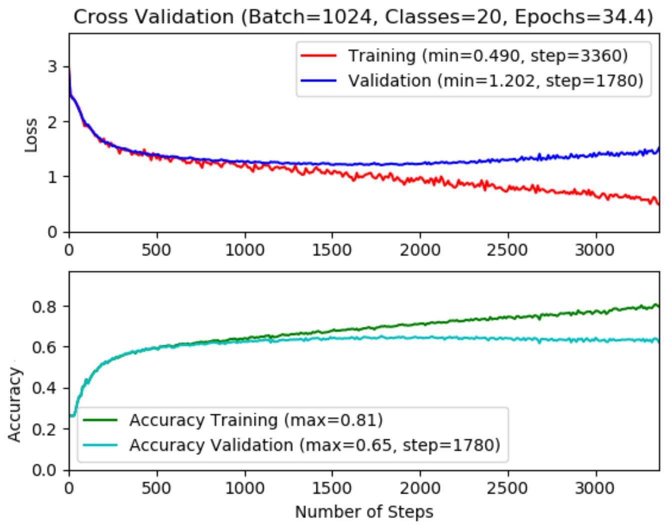

Figure 195 Cross validation aiming at 20 prepositional classes

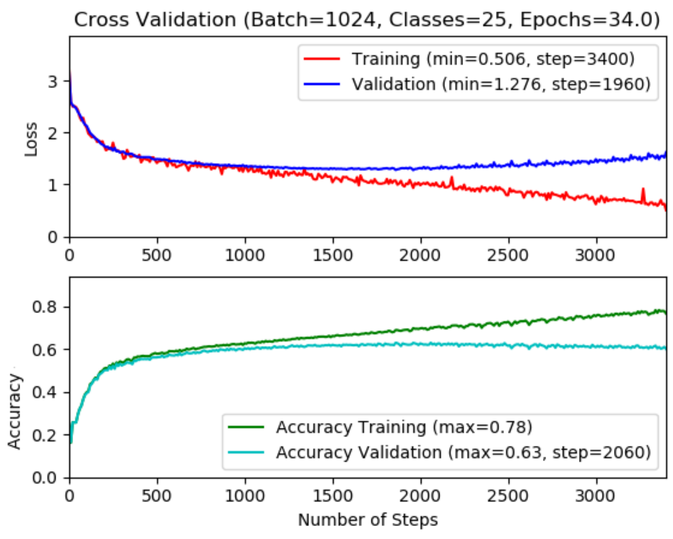

Figure 196 Cross validation aiming at 25 prepositional classes

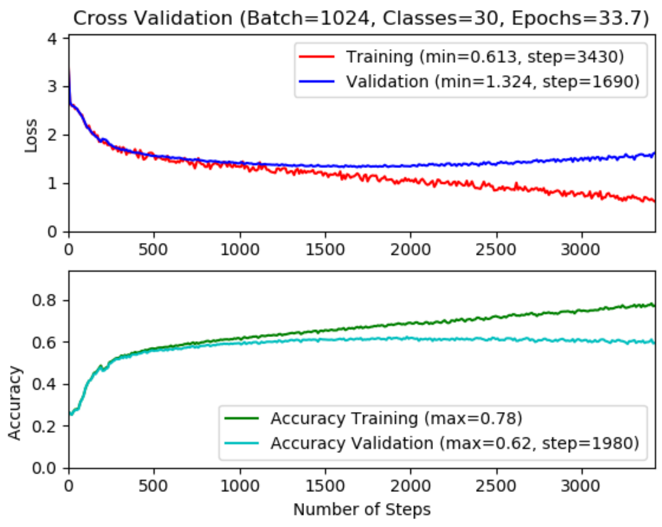

Figure 197 Cross validation aiming at 30 prepositional classes

4.6.4.2 Testing Phase

4.6.4.2.1 Precision-Recall Graphs

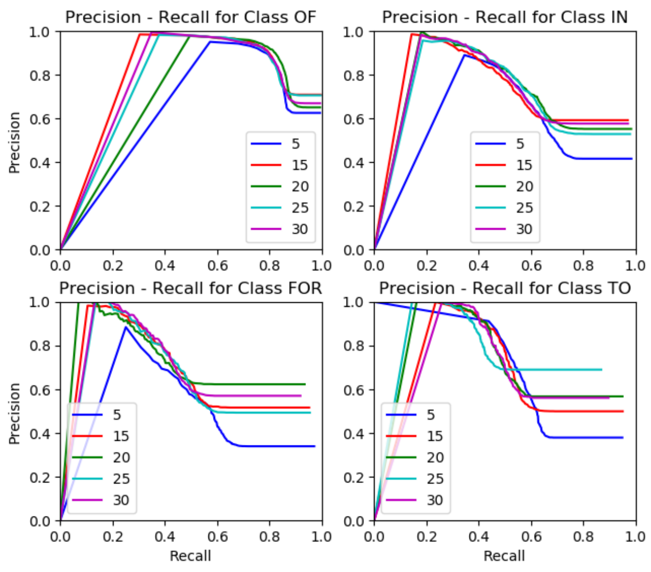

Figure 198 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the number of prepositional classes

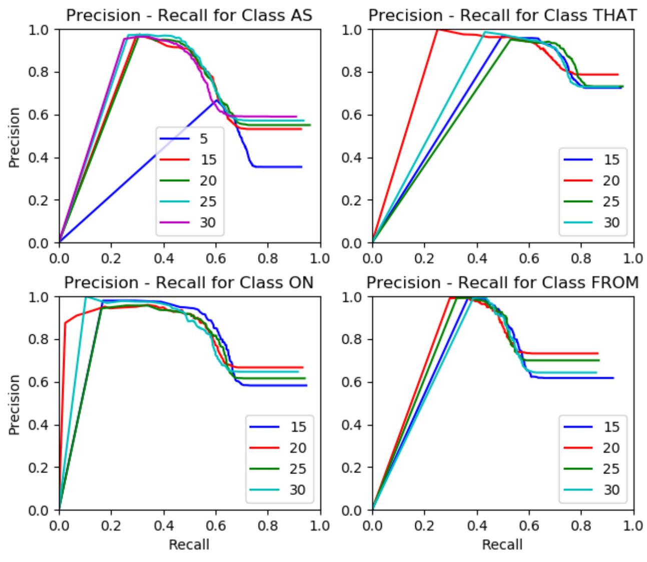

Figure 199 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the number of prepositional classes

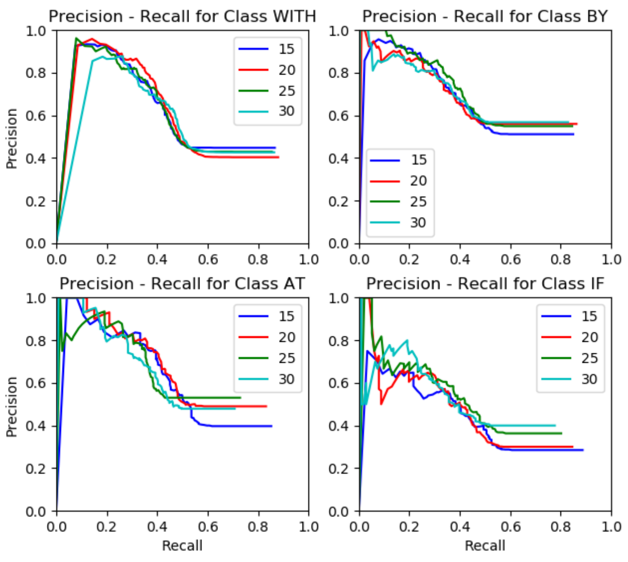

Figure 200 Precision vs. Recall for classes WITH (upper left) and BY (upper right), AT (lower left) and IF (lower right) varying the number of prepositional classes

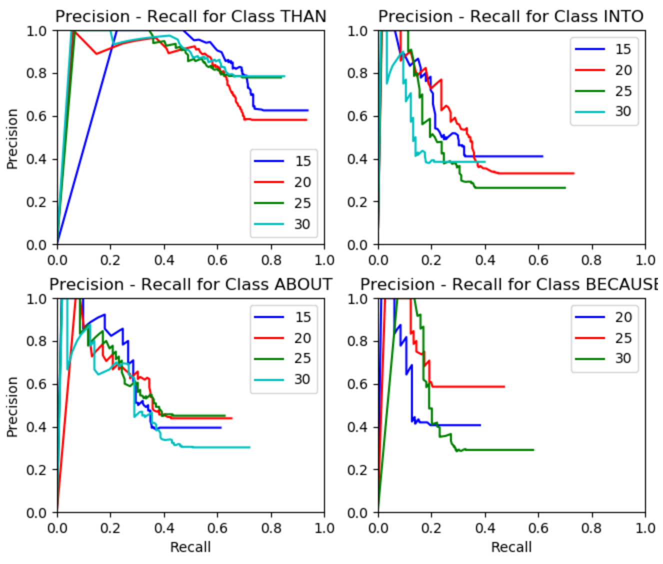

Figure 201 Precision vs. Recall for classes THAN (upper left) and INTO (upper right), ABOUT (lower left) and BECAUSE (lower right) varying the number of prepositional classes

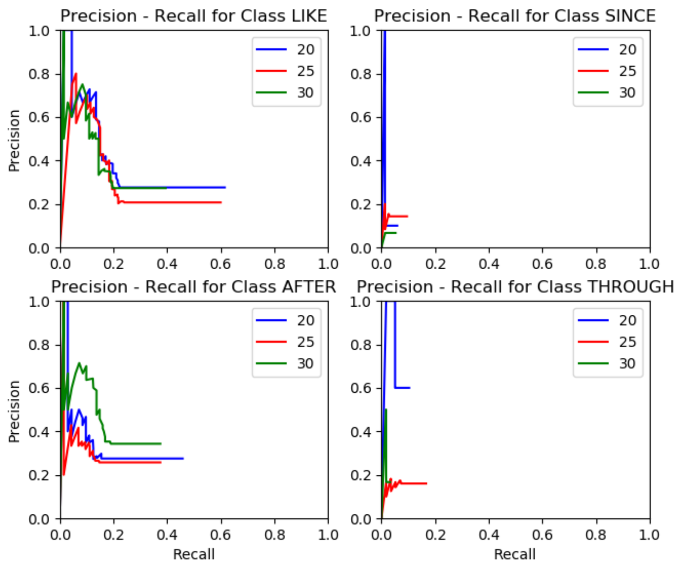

Figure 202 Precision vs. Recall for classes LIKE (upper left), SINCE (upper right), AFTER (lower left) and THROUGH (lower right) varying the number of prepositional classes

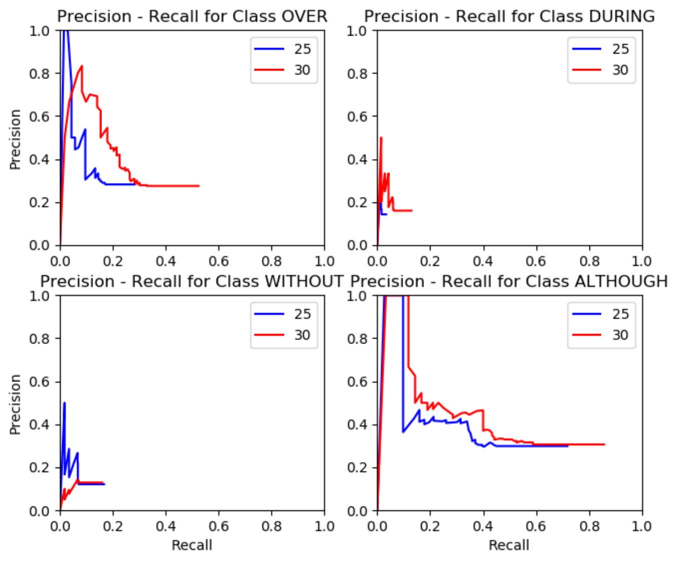

Figure 203 Precision vs. Recall for classes OVER (upper left), DURING (upper right), WITHOUT (lower left) and ALTHOUGH (lower right) varying the number of prepositional classes

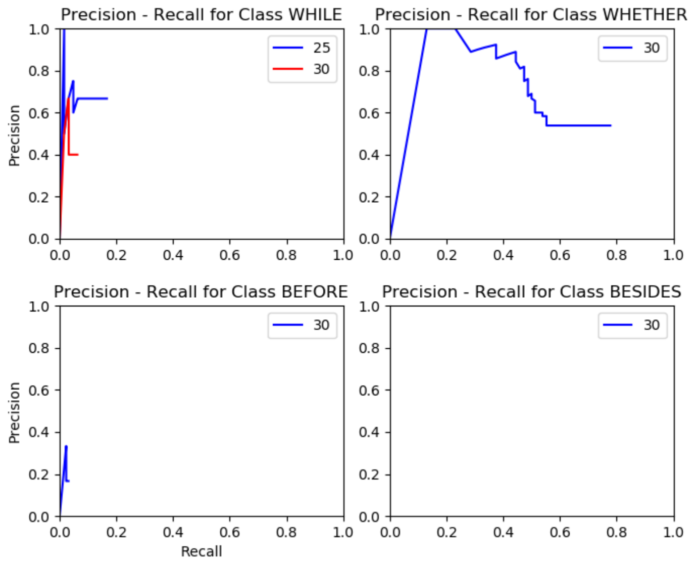

Figure 204 Precision vs. Recall for classes WHILE (upper left), WHETHER (upper right), BEFORE (lower left) and BESIDES (lower right) varying the number of prepositional classes

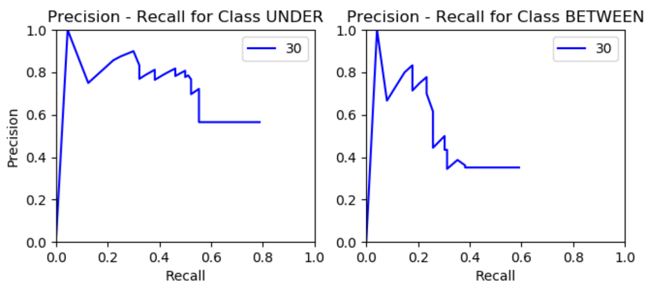

Figure 205 Precision vs. Recall for classes UNDER (upper left) and BETWEEN (upper right) varying the number of prepositional classes

4.6.4.2.2 ROC Graphs

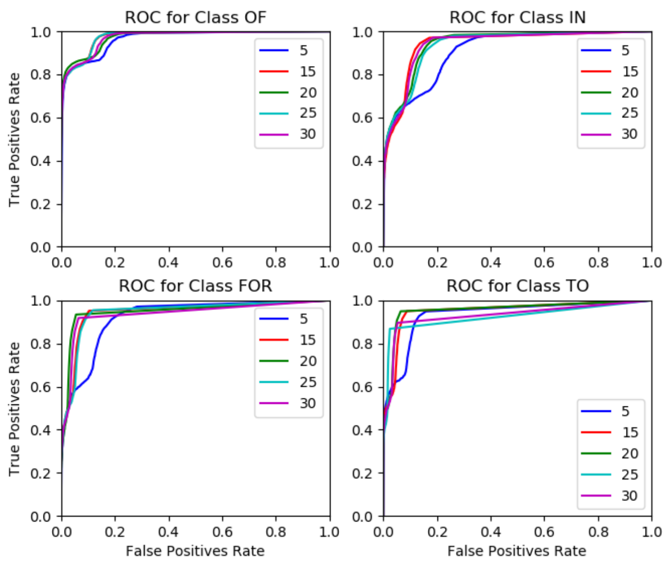

Figure 206 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) varying the number of prepositional classes

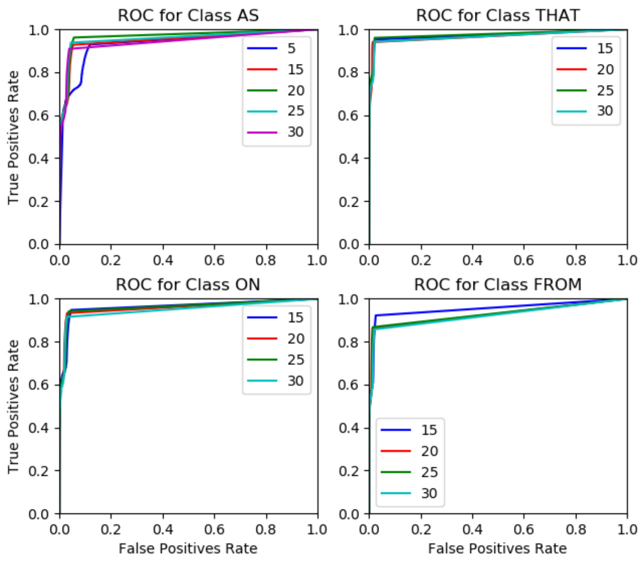

Figure 207 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) varying the number of prepositional classes

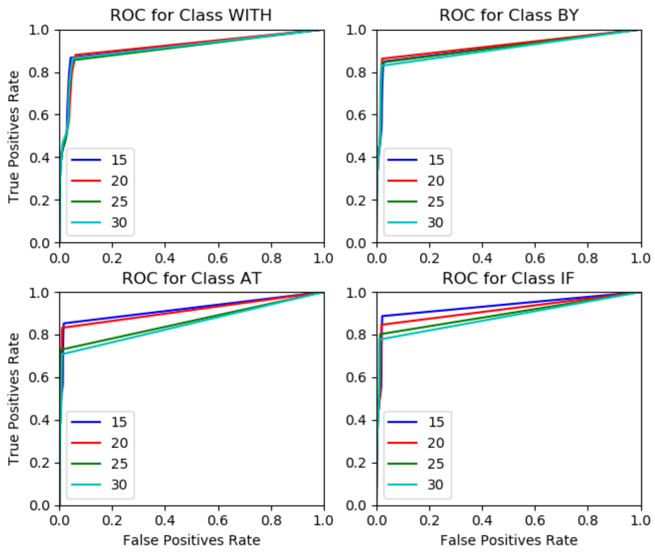

Figure 208 for classes WITH (upper left), BY (upper right), AT (lower left) and IF (lower right) varying the number of prepositional classes

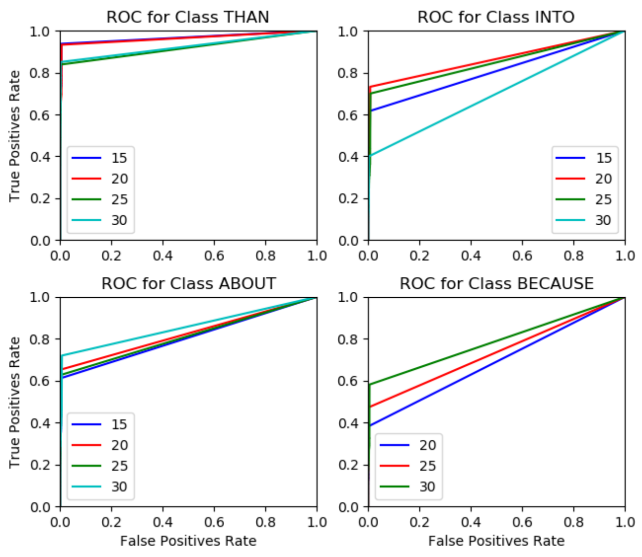

Figure 209 ROC for classes THAN (upper left), INTO (upper right), ABOUT (lower left) and BECAUSE (lower right) varying the number of prepositional classes

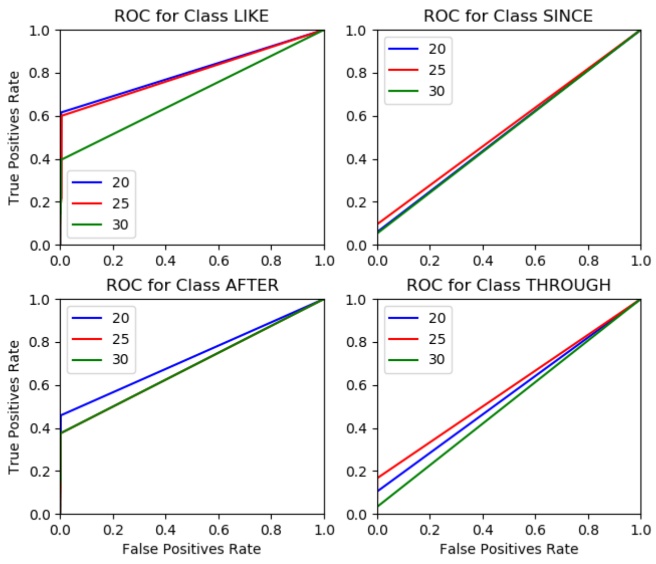

Figure 210 ROC for classes LIKE (upper left), SINCE (upper right), AFTER (lower left) and THROUGH (lower right) varying the number of prepositional classes

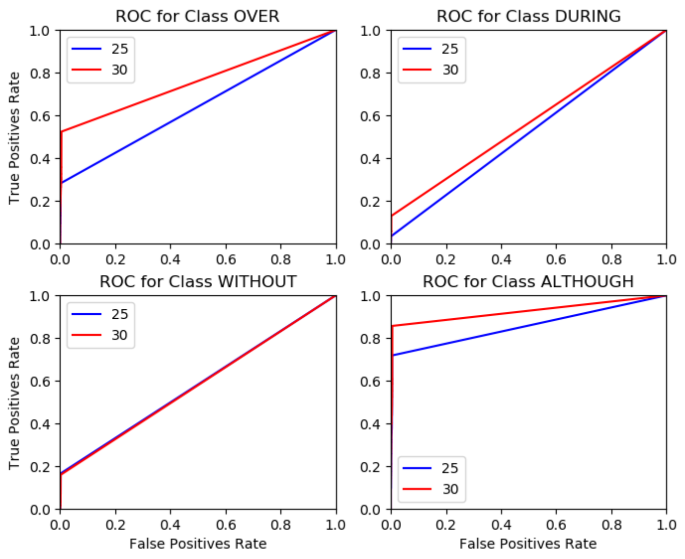

Figure 211 ROC for classes OVER (upper left), DURING (upper right), WITHOUT (lower left) and ALTHOUGH (lower right) varying the number of prepositional classes

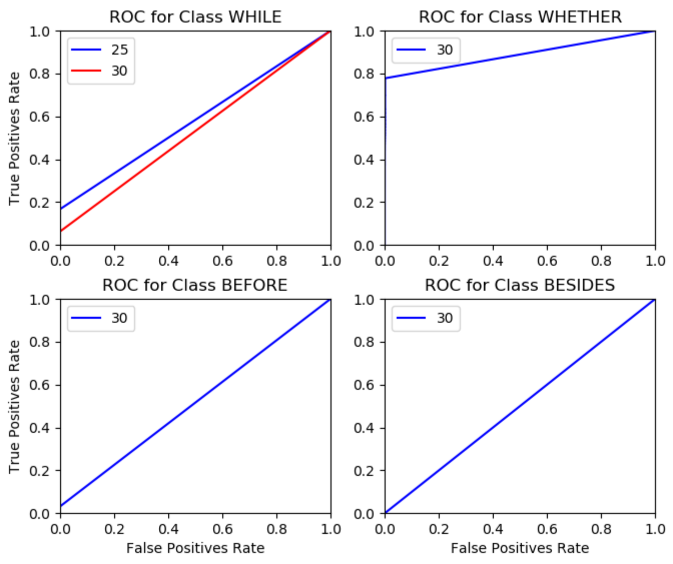

Figure 212 ROC for classes WHILE (upper left), WHETHER (upper right), BEFORE (lower left) and BESIDES (lower right) varying the number of prepositional classes

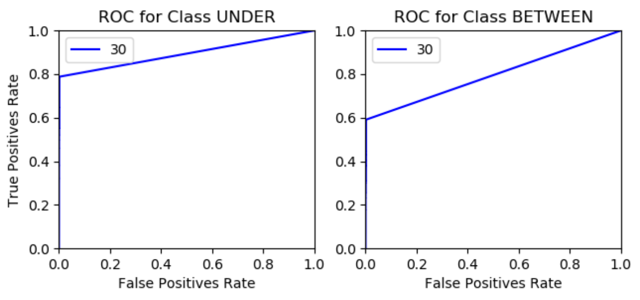

Figure 213 ROC for classes UNDER (left) and BETWEEN (right) varying the number of prepositional classes

4.6.4.2.3 Metric for Error Correction against a Threshold

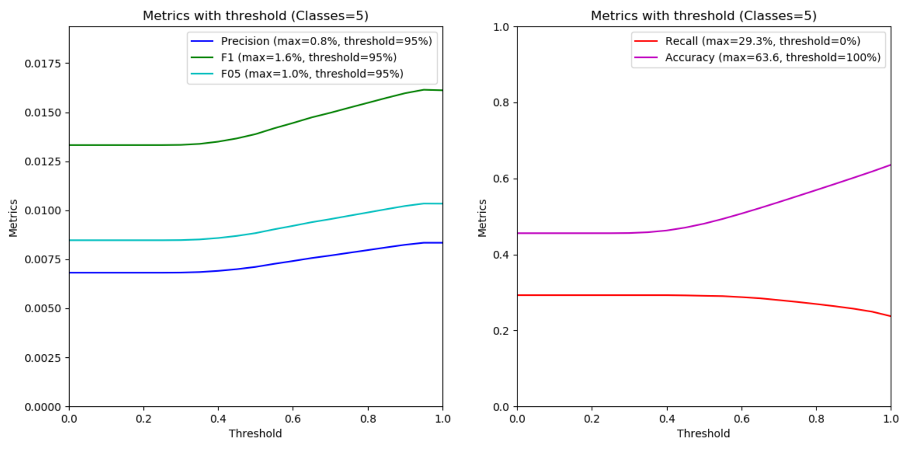

Figure 214 Correction Metrics against a Threshold aiming at 5 prepositional classes

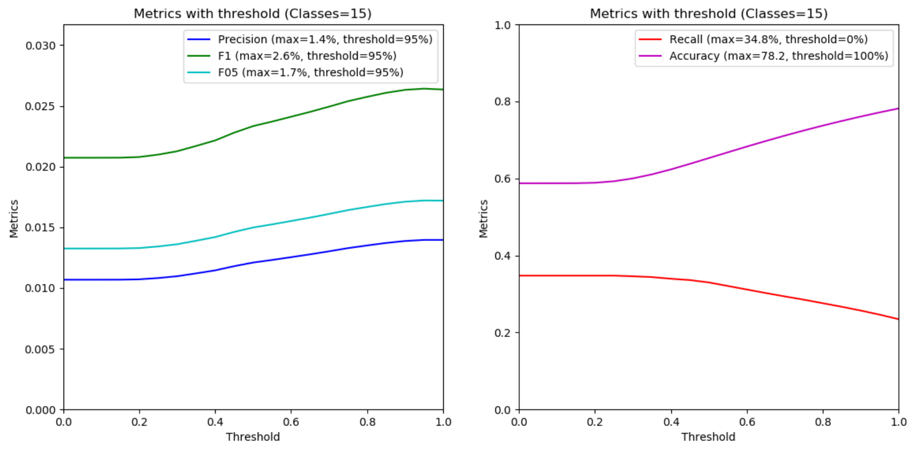

Figure 215 Correction Metrics against a Threshold aiming at 15 prepositional classes

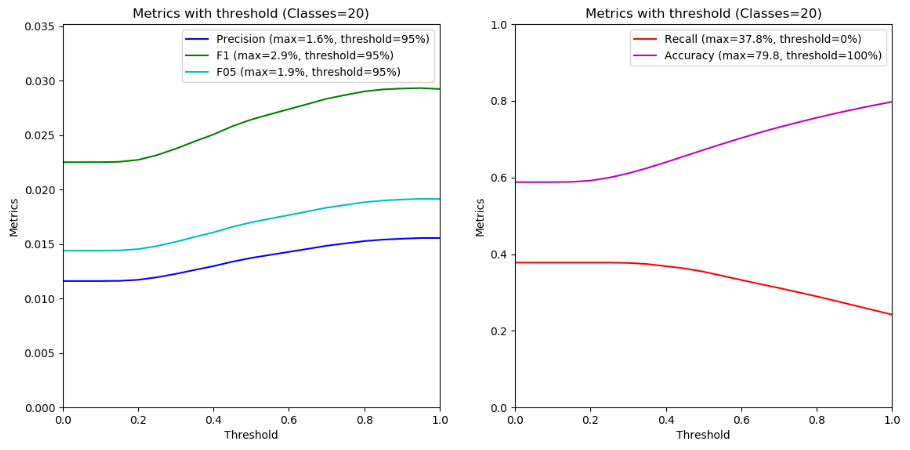

Figure 216 Correction Metrics against a Threshold aiming at 20 prepositional classes

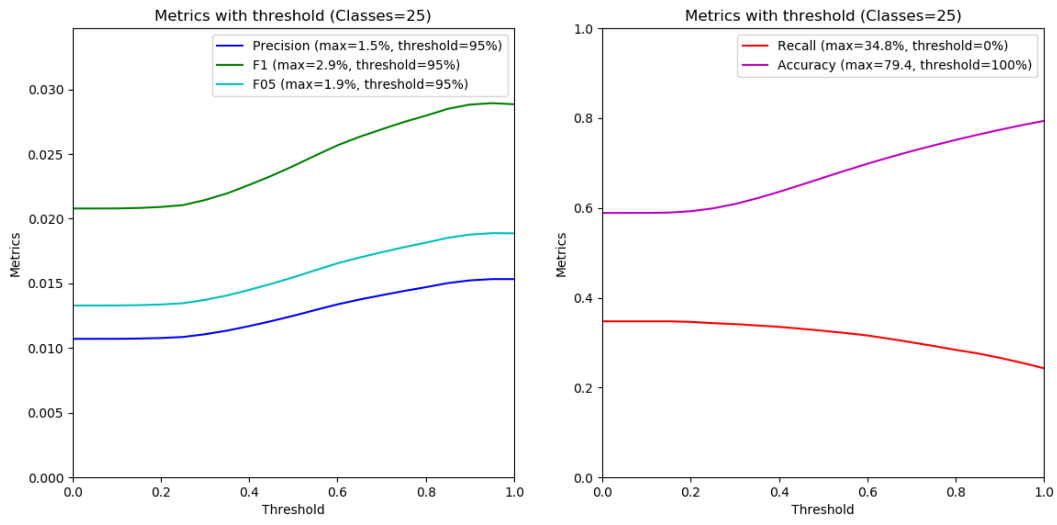

Figure 217 Correction Metrics against a Threshold aiming at 25 prepositional classes

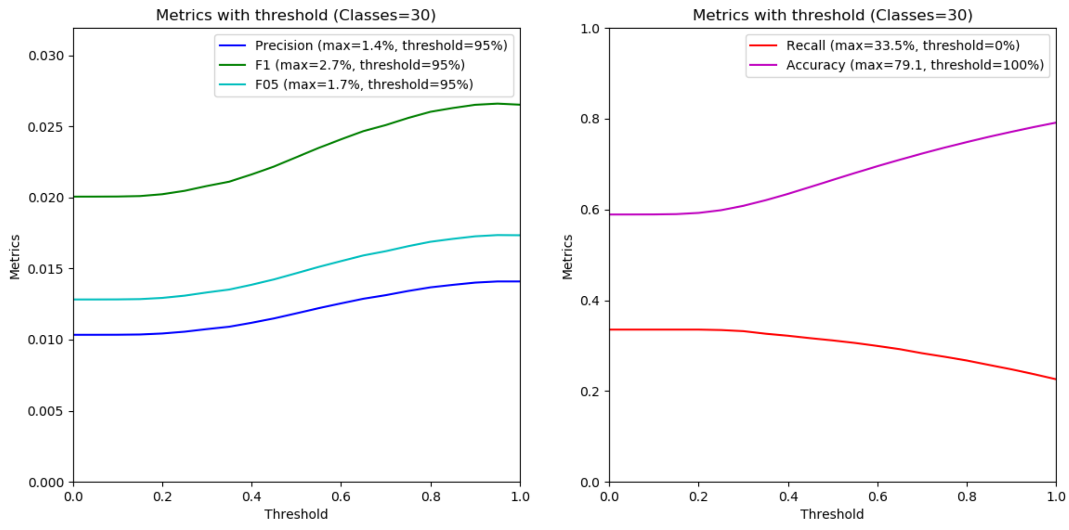

Figure 218 Correction Metrics against a Threshold aiming at 30 prepositional classes

4.6.4.2.4 Metrics for Error Detection and Correction Tasks

Classes	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
5	0.68%	29.26%	1.33%	0.85%	45.58%
10	1.10%	37.8%	2.14%	1.37%	56.64%
15	1.07%	34.76%	2.07%	1.32%	58.75%
20	1.16%	37.80%	2.25%	1.44%	58.79%
25	1.07%	34.76%	2.08%	1.33%	58.90%
30	1.03%	33.54%	2.01%	1.28%	58.90%

Table 66 Error correction for all classes varying the number of prepositional classes

Classes	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
5	1.69%	72.56%	3.30%	2.10%	46.37%
10	2.4%	82.32%	4.66%	2.98%	57.53%
15	2.27%	73.78%	4.40%	2.81%	59.54%
20	2.45%	79.88%	4.75%	3.04%	59.64%
25	2.42%	78.66%	4.70%	3.01%	59.78%
30	2.48%	80.49%	4.81%	3.08%	59.84%

Table 67 Error detection for all classes varying the number of prepositional classes

Classes	Precision	Recall	F1	F05	Accuracy
5	2.50%	47.52%	4.76%	3.09%	75.62%
10	1.73%	45.25%	3.33%	2.14%	67.27%
15	1.42%	37.01%	2.74%	1.76%	65.49%
20	1.41%	40.00%	2.73%	1.75%	63.46%
25	1.21%	36.07%	2.35%	1.50%	61.87%
30	1.12%	34.81%	2.17%	1.39%	60.85%

Table 68 Error correction for only the target classes varying the number of prepositional classes

Classes	Precision	Recall	F1	F05	Accuracy
5	4.33%	82.18%	8.23%	5.35%	76.40%
10	3.34%	87.59%	6.44%	4.13%	68.15%
15	2.80%	72.72%	5.39%	3.46%	66.27%
20	2.85%	80.64%	5.50%	3.53%	64.32%
25	2.61%	77.84%	5.06%	3.24%	62.74%
30	2.56%	79.75%	4.98%	3.19%	61.76%

Table 69 Error detection for only the target classes varying the number of prepositional classes

4.7 Pre-processing

4.7.1 Spelling

4.7.1.1 Training Phase

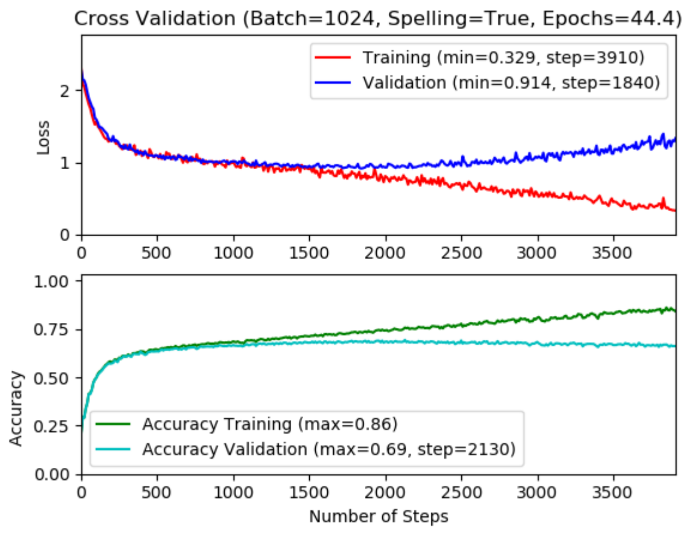

Figure 219 Cross validation correcting spelling

4.7.1.2 Testing Phase

4.7.1.2.1 Precision-Recall Graphs

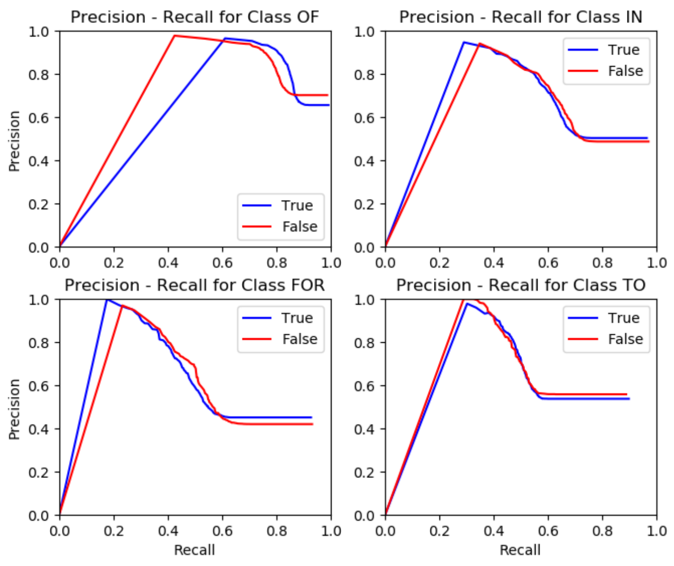

Figure 220 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) comparing spelling correction

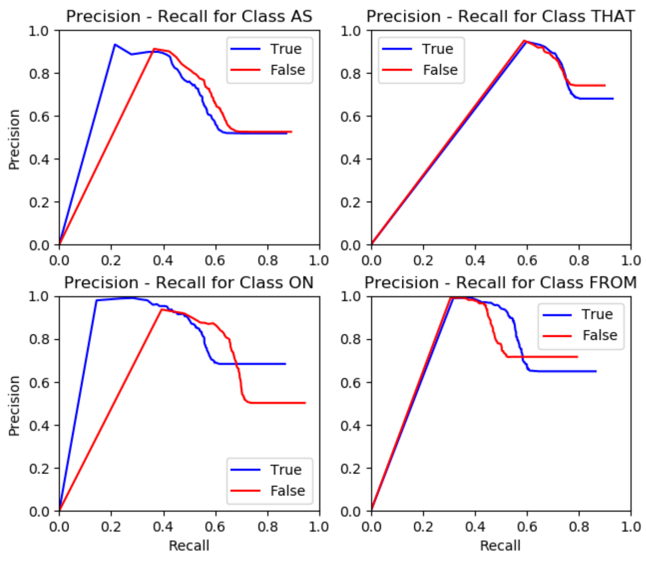

Figure 221 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) comparing spelling correction

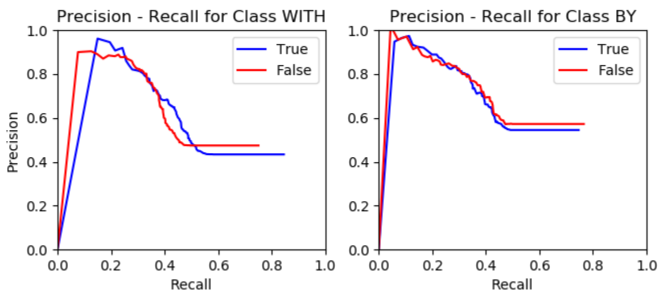

Figure 222 Precision vs. Recall for classes WITH (left) and BY (right) comparing spelling correction

4.7.1.2.2 ROC Graphs

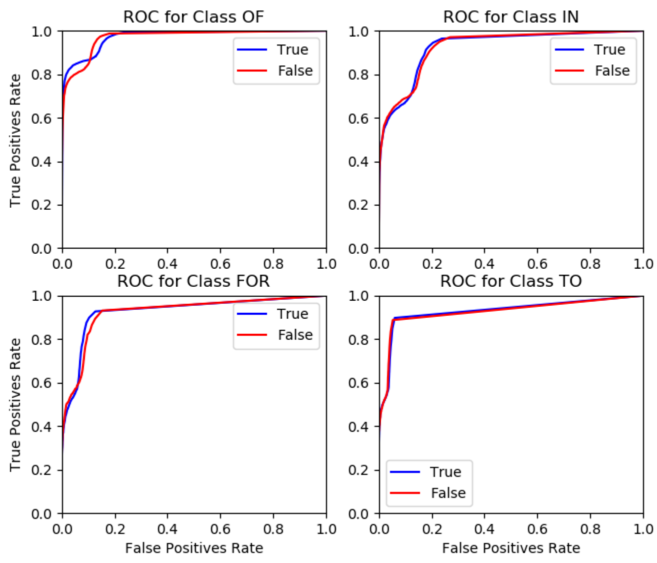

Figure 223 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) comparing spelling correction

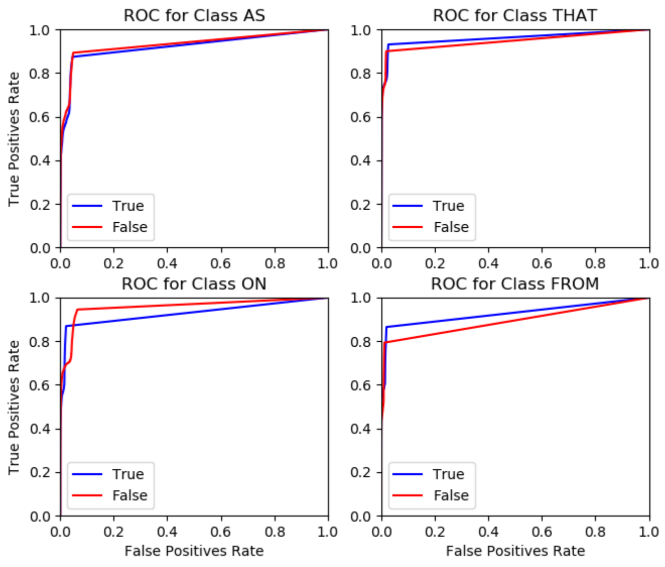

Figure 224 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) comparing spelling correction

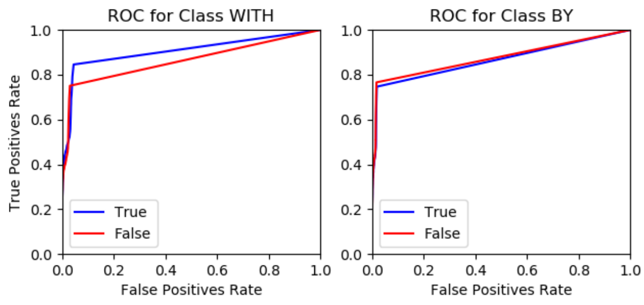

Figure 225 ROC for classes WITH (left) and BY (right) comparing spelling correction

4.7.1.2.3 Metrics for Error Correction against a Threshold

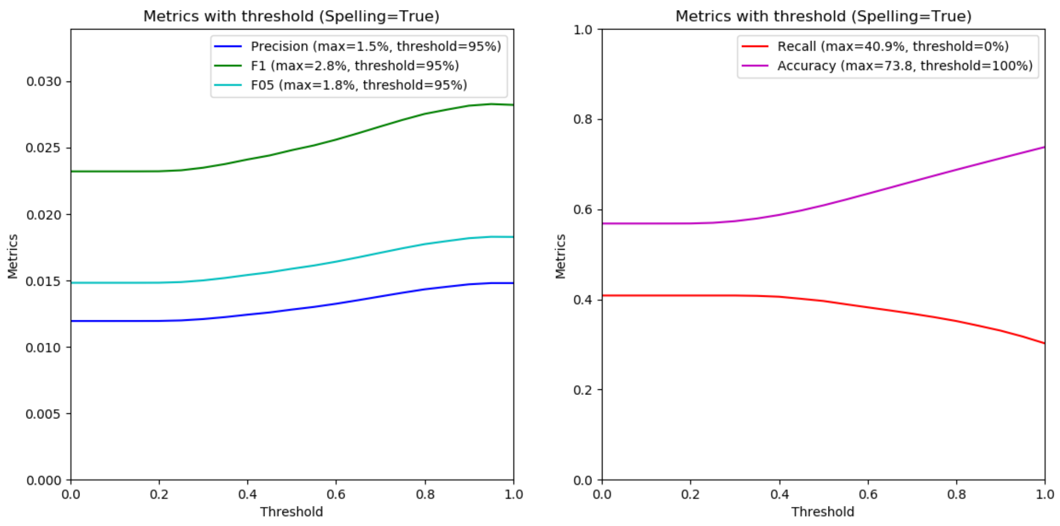

Figure 226 Correction Metrics against a Threshold correcting spelling

4.7.1.2.4 Metrics for Error Detection and Correction Tasks

Spelling	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
True	1.19%	40.85%	2.32%	1.48%	56.83%
False	1.10%	37.8%	2.14%	1.37%	56.64%

Table 70 Error correction for all classes comparing spelling correction

Spelling	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
True	2.39%	81.70%	4.64%	2.96%	57.64%
False	2.4%	82.32%	4.66%	2.98%	57.53%

Table 71 Error detection for all classes comparing spelling correction

Spelling	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
True	1.88%	48.90%	3.61%	2.32%	67.50%
False	1.73%	45.25%	3.33%	2.14%	67.27%

Table 72 Error correction for 10 top classes comparing spelling correction

Spelling	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
True	3.25%	84.67%	6.26%	4.02%	68.26%
False	3.34%	87.59%	6.44%	4.13%	68.15%

Table 73 Error detection for 10 top classes comparing spelling correction

4.8 Optimum Algorithm and Final Algorithm

The optimum algorithm uses the optimum values achieved through all other experiments and which are summarized in table 75. The final algorithm is the optimum algorithm using tuned thresholds.

Parameter	**Default**
Hidden Cells	1600
Number of layers	2
Model	simple
Input Type	embedding
Word Embedding Size	100
Embedding Window	1
Number of Classes	20
Spelling	True

POS Tags	True
Tags Embedding Size	15
Index of Parent in Dependency Tree	True
Index Embedding Size	10
Lower Limit for Vocabulary	2
Oversampling	False
Oversampling Position	0
Undersampling	False
Undersampling Position	0
Length of Sequence Input (at each side)	5
Original Preposition	replace
Dropout	False
Attention	True

Table 74 Optimum parameters of the algorithm

4.8.1 Training Phase

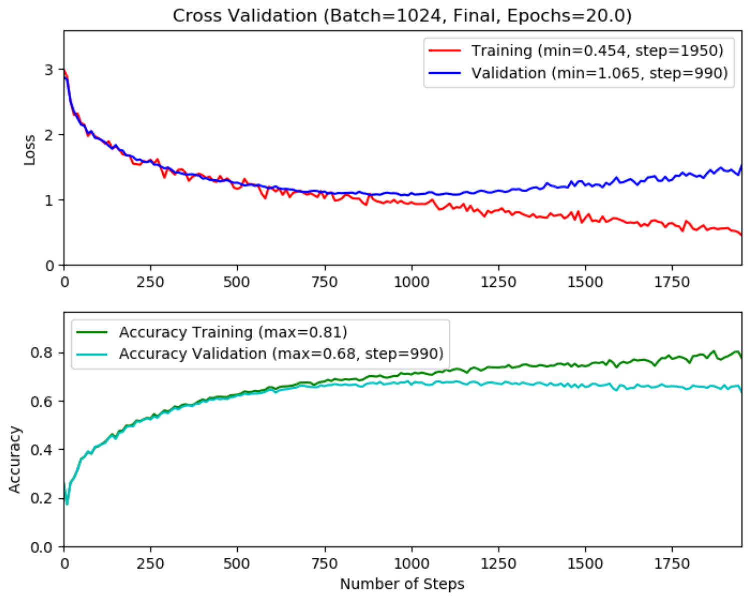

Figure 227 Cross validation using optimum parameters

4.8.2 Testing Phase

4.8.2.1 Precision-Recall Graphs

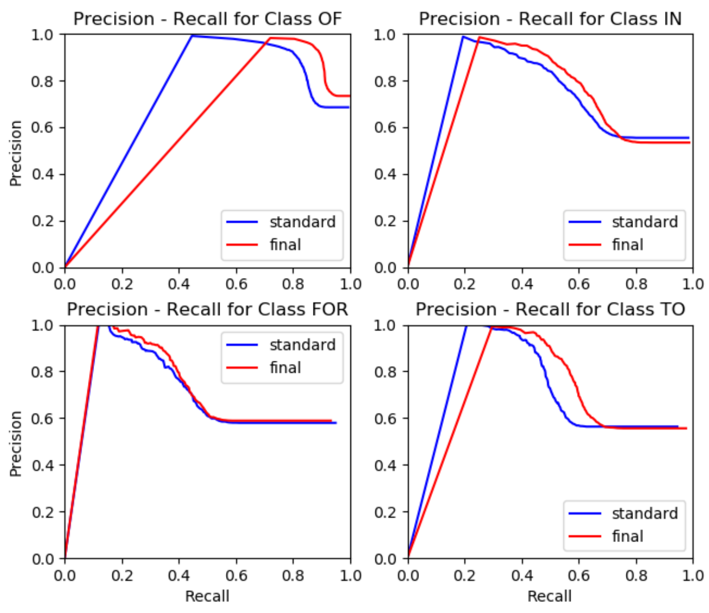

Figure 228 Precision vs. Recall for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) optimum vs. standard

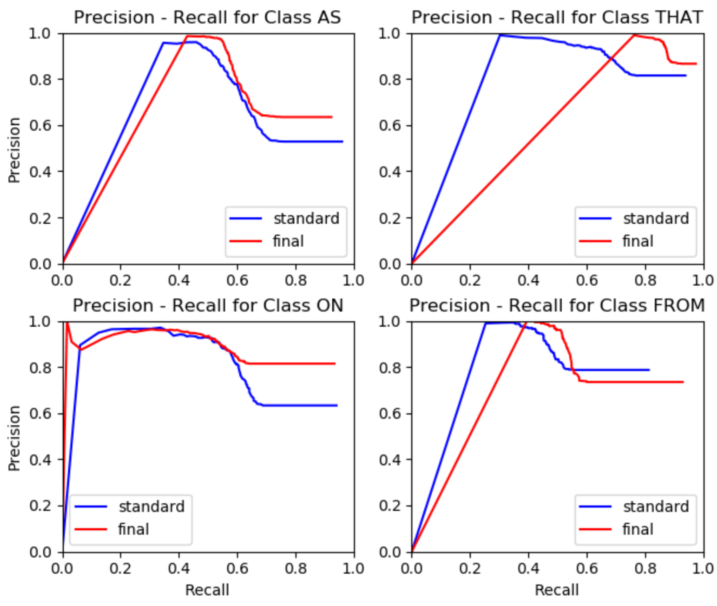

Figure 229 Precision vs. Recall for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) optimum vs. standard

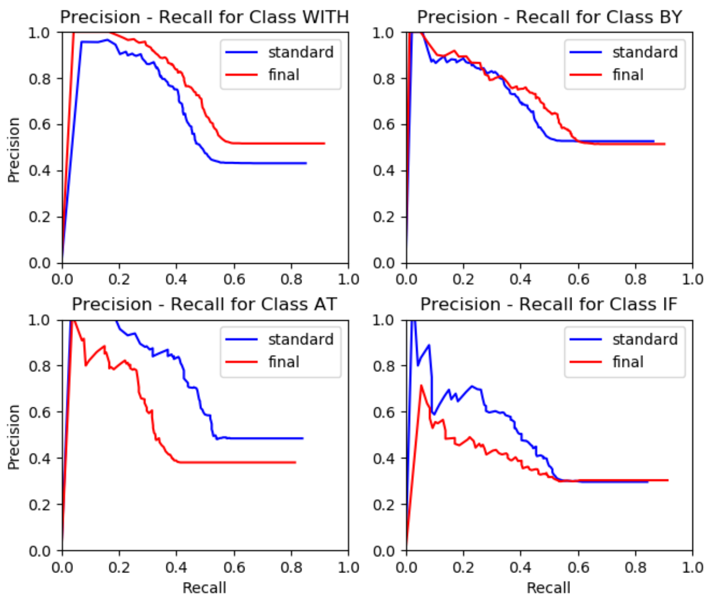

Figure 230 Precision vs. Recall for classes WITH (upper left), BY (upper right), AT (lower left) and IF (lower right) optimum vs. standard

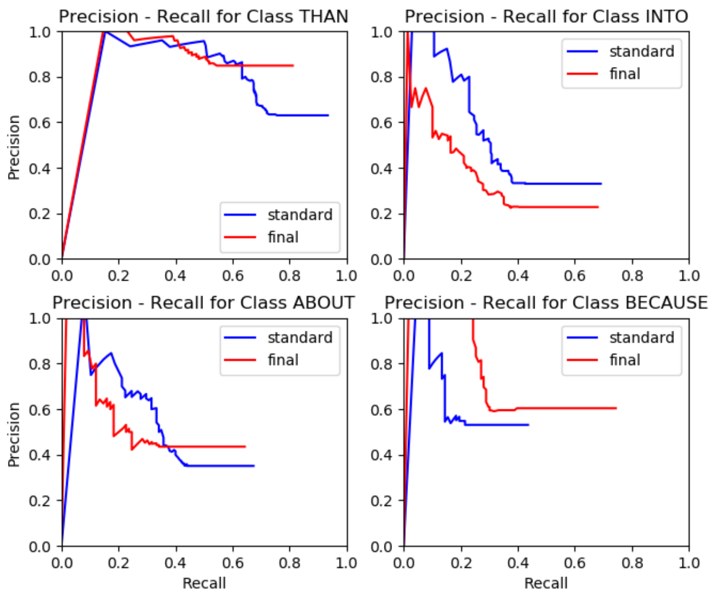

Figure 231 Precision vs. Recall for classes THAN (upper left), INTO (upper right), ABOUT (lower left) and BECAUSE (lower right) optimum vs. standard

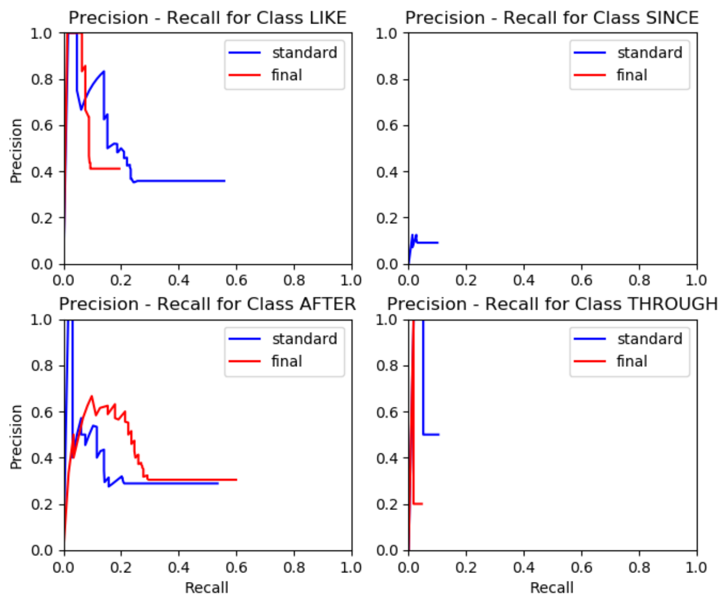

Figure 232 Precision vs. Recall for classes LIKE (upper left), SINCE (upper right), AFTER (lower left) and THROUGH (lower right) optimum vs. standard

4.8.2.2 ROC Graphs

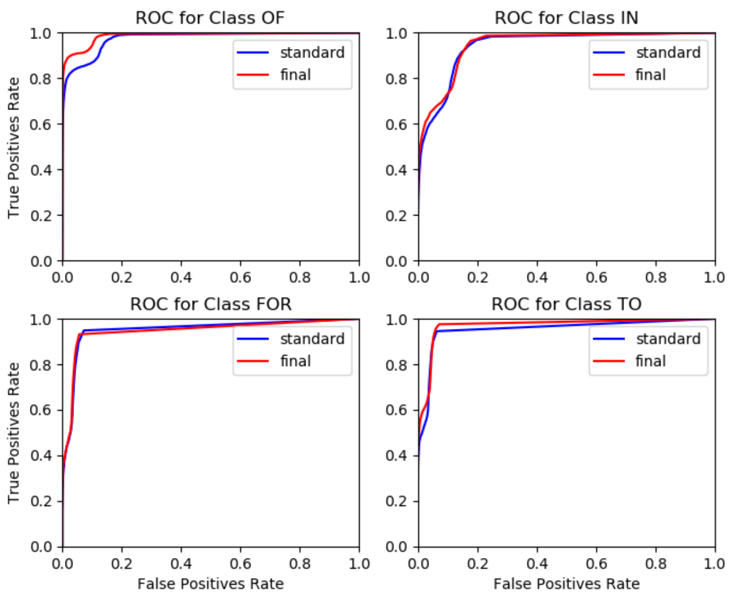

Figure 233 ROC for classes OF (upper left), IN (upper right), FOR (lower left) and TO (lower right) optimum vs. standard

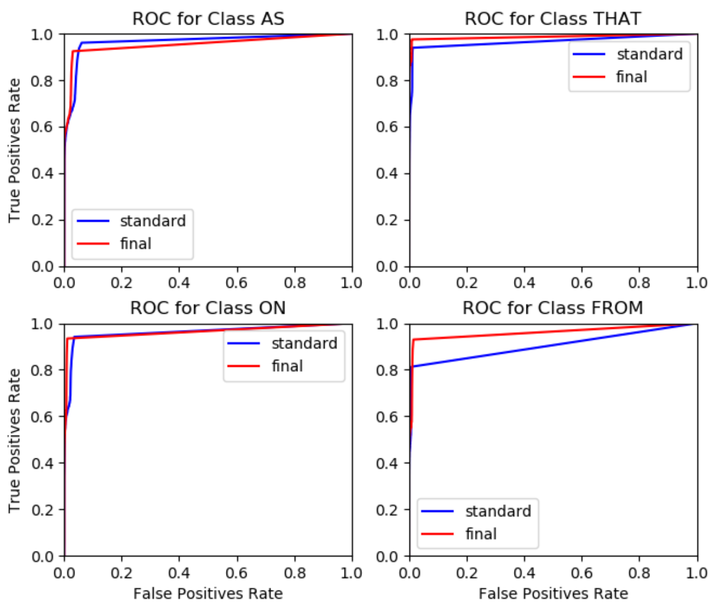

Figure 234 ROC for classes AS (upper left), THAT (upper right), ON (lower left) and FROM (lower right) optimum vs. standard

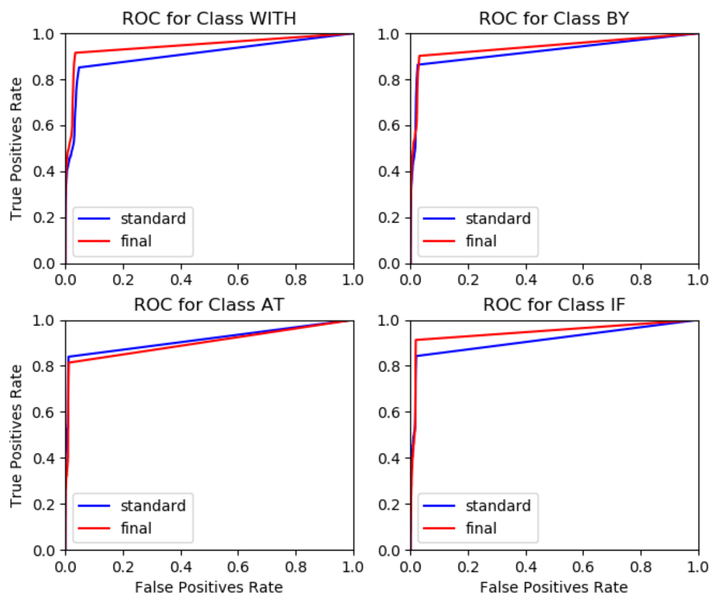

Figure 235 ROC for classes WITH (upper left), BY (upper right), AT (lower left) and IF (lower right) optimum vs. standard

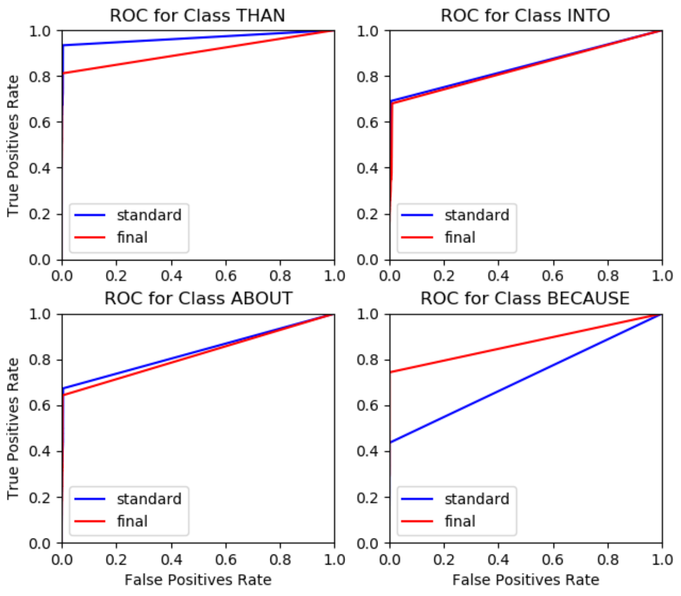

Figure 236 ROC for classes THAN (upper left), INTO (upper right), ABOUT (lower left) and BECAUSE (lower right) optimum vs. standard

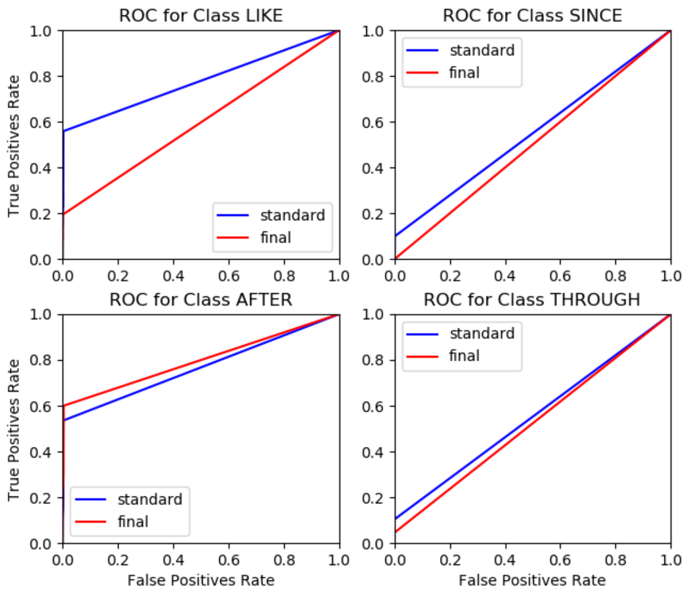

Figure 237 ROC for classes LIKE (upper left), SINCE (upper right), AFTER (lower left) and THROUGH (lower right) optimum vs. standard

4.8.2.3 *Metrics for Error Correction against a Threshold*

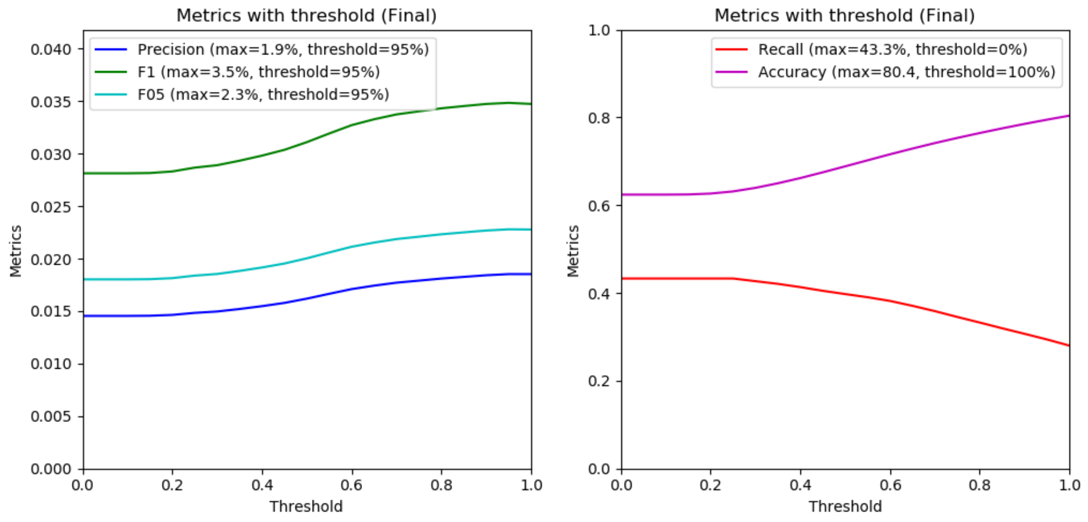

Figure 238 Correction Metrics against a Threshold using optimum parameters

4.8.2.4 *Metrics for Error Detection and Correction Tasks*

Once the optimum algorithm predicts a preposition, the final algorithm applies a tuned threshold per class (which was deduced from all previous experiments). For the classes “of”, “that”, “on”, “from” and “than”, which are the ones that the algorithm best recognizes, the threshold is 80%; for classes “in”, “for”, “to”, “as”, “with”, “by” and “because”, which are somehow separated, the threshold is 90%; and for all other classes, which are not well recognized, the threshold is 100.0% (the final algorithm does not rely on these prepositions).

Algorithm	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
Optimum	1.45%	43.29%	2.81%	1.80%	62.41%
Standard	1.16%	37.80%	2.25%	1.44%	58.79%
Final Algorithm	4.40%	10.36%	6.18%	4.97%	96.03%

Table 75 Error correction for all classes optimum vs. standard vs. final algorithm

Algorithm	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
Optimum	2.66%	79.26%	5.15%	3.30%	63.15%
Standard	2.45%	79.88%	4.75%	3.04%	59.64%
Final Algorithm	5.44%	12.80%	7.63%	6.15%	96.09%

Table 76 Error detection for all classes optimum vs. standard vs. final algorithm

Algorithm	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
Optimum	1.80%	45.80%	3.45%	2.23%	67.38%
Standard	1.41%	40.00%	2.73%	1.75%	63.46%
Final Algorithm	5.12%	10.97%	6.98%	5.73%	96.24%

Table 77 Error correction for top 20 classes optimum vs. standard vs. final algorithm

Algorithm	**Precision**	**Recall**	**F1**	**F05**	**Accuracy**
Optimum	3.12%	79.35%	6.02%	3.87%	68.10%
Standard	2.85%	80.64%	5.50%	3.53%	64.32%
Final Algorithm	6.32%	13.55%	8.62%	7.08%	96.30%

Table 78 Error correction for 20 top classes optimum vs. standard vs. final algorithm

4.8.2.5 Comparison of the Final Algorithm against Teams of the Shared Task CoNLL 2013

Team	**Precision**	**Recall**	**F1**	External Resources
NARA	29.10	12.54	17.53	Lang-8
STEL	25.66	9.32	13.68	Wikipedia, WordNet
NTHU	12.01	12.86	12.42	Google Web-1T
UIUC	26.53	4.18	7.22	Google Web-1T, Gigaword
TILB	5.07	10.61	6.86	Google Web-1T, Gigaword
CAMB	40.74	3.54	6.51	Cambridge Learner Corpus
Final Algorithm*	11.29 (11.29)	4.30 (10.85)	6.24 (11.07)	None
HIT	28.12	2.89	5.25	WordNet, Longman dictionary
UMC	35.29	1.93	3.66	News corpus, JMySpell dictionary, Google Web-1T, Penn Treebank
TOR	5.38	2.25	3.17	Wikipedia
SJT1	12.50	1.29	2.33	Europarl
STAN	20.00	0.32	0.63	English Resource Grammar
KOR	4.76	0.32	0.60	None
UAB	0.00	0.00	0.00	Top 250 uncountable nouns, FreeLing morphological dictionary
SJT2	0.00	0.00	0.00	-

*Table 79 Final algorithm compared to shared task CoNLL 2013. *the final algorithm only deals with prepositional wrong choice errors, whose isolated value is shown between parentheses*

5 Analysis
From the results, three patterns can be highlighted: first, the low number of errors affects the precision metric significantly, reducing significantly its value; second, the relatively large number of true negatives increases the accuracy as the threshold increases, which is quite natural if one takes into account the low rate of prepositional errors in the corpus, i.e. if the algorithm flags all original prepositions as correctly used, the accuracy would achieve a value of 98.99% while the metrics for the error correction task would be zero. Finally, as the threshold of decision increases, the metrics for the task of prepositional error detection and correction improves on performances (metrics for error detection and correction against a threshold). In the next section, the analysis of each experiment is described in more detail.

5.1 Size of the Neural Network
For the number of hidden cells, the training graphs (figures 5 - 10) shows that as the number of hidden cells increases the training process requires fewer epochs to train the algorithm, but at the same time the loss rate becomes less stable and the training dataset starts to overfit more quickly. Moreover, while the number of hidden cells increases the loss function tends to decrease slightly, meaning that a bigger number of hidden cells can improve the performance of the algorithm in a moderate way. This is somewhat confirmed by the values of the metrics for the error detection and correction presented in tables 6, 7, 8 and 9 where all metrics increase slightly while the number of cells increases with some drops at values of hidden cells of 200 and 800. Therefore, the evidence suggests that the algorithm performs the best for the task of prepositional error detection and correction when using 1600 hidden cells, which makes sense as a larger number of hidden cells can keep the recurrent neural network more informed in order to make predictions.

Moreover, analysing the Precision-Recall graphs as well as the ROC graphs (figures 11 - 16), one can see that for the classes 'of' and 'that', the algorithm performs well for all values, in which a high recall can be obtained without losing much precision. For the class *"of"*, this can be expected as this class contains the larger number of examples so that the algorithm can learn to distinguish this class better than all others. However, the class *"that"* is distinguished in a more precise way than the classes "in", "for", and "to", which present a larger number of instances in the corpus, indicating that some prepositions (or conjunctions and adverbs) can be more easily identified by the algorithm given some lexical features than other classes. The difficulty of mastering some types of prepositions applies as well for English learners as it is somewhat confirmed by the difference in the rate of errors per class given in table 4, in which the error rate for classes such as "for" and "to" is much larger than the error rate for classes such as "that" and "from". The prepositions "from", "as" and "to" are separated reasonably, while for the prepositions "by" and "with", the precision decreases rapidly as the recall increases, meaning that on the one hand, the confidence of the algorithm to predict these classes should be set at a high value and, on the other hand, the number of errors for these classes is more difficult to identify. The classes "on", "in" and "for" are slightly separated, meaning that the threshold for these classes should be high. Finally, the capacity of the algorithm to separate all target classes is similar for all the different numbers of cells.

On the other hand, the increment in the number of layers does not show any improvement on the performance of the algorithm. Contrarily, the metrics for the task of detecting and correcting errors (tables 10, 11, 12 and 13) show a decrement on performance as the number of layers increases, with a rise for a number of layers of 2. This behaviour can be noticed as well on the experiments related to dropout in which the deepest neural networks tested in this study did not perform better than any other model. Moreover, the training graphs (figures 23 - 25) show that the loss function does not fall using a larger number of layers but the algorithm needs more epochs to be trained. Furthermore, precision-recall graphs and ROC graphs (figures 26 - 31) do not show a different behaviour than that already seen for the hidden cells experiments, showing that no particular class is affected by the rise in the size of the neural network, but that the separation may rather respond to the size of the training data or another hidden parameter not detected yet. The poor results using deep neural networks for correcting prepositional errors goes against the results of deep recurrent networks in other areas (Graves, Mohamed & Hinton 2013; Hermans & Schrauwen 2013), but at the same time proves that the use of deep recurrent networks needs further investigation to bring about the full benefits of its use. Finally, for the scope of this study, the final neural network was built using two layers, which rendered the best results.

5.2 Data

Unlike the result obtained in the experiments to define the size of the neural network, the experiments to compensate for the imbalanced nature of the training data using oversampling produced conclusive results. On the one hand, figures 35, 36, 37 and 38 show that the classifier loses performance as the training data is undersampled further. It is also confirmed by the metrics given in tables 14, 15, 16 and 17 in which the performance of the algorithm falls while the number of examples is truncated. As expected, the classes that are more deeply affected are the ones with the larger number of examples. For instance, in figures 39 and 42, the classes "of", "in", "for" and "to" start to be less identifiable as the parameter "—under_pos" increases, i.e. as the number of examples is reduced for these classes. Moreover, oversampling also has a negative effect on classes with smaller numbers of examples but in a less severe manner. This could be caused by the fact that if the classifier cannot properly distinguish some classes (e.g. top classes), then it also becomes harder to distinguish classes with fewer occurrences.

On the other hand, although the experiments on oversampling were less conclusive, one could say that oversampling is not a good way to compensate for the imbalanced number of occurrences for each class. Either the behaviour of the algorithm to separate the classes did not show any difference or improvement on the classes with fewer occurrences (figures 54, 55, 57 and 58), but, on the contrary, when compared against the algorithm trained without using oversampling (tables 18, 19, 20 and 21), the algorithms trained on oversampled examples perform worse. Therefore, neither undersampling nor oversampling are appropriate methods to compensate for the imbalanced number of examples in a corpus when training an algorithm to detect and correct prepositional errors, but rather the usage of the data as it is (imbalanced) is a more effective method.

5.3 Examples

The LSTM architecture was designed to solve the long dependency problem, but the experiments performed in this study varying the maximum length of each example showed that a larger length of the input sequence causes the algorithm to perform worse. This tendency is easily identifiable in the training graphs (figure 63, 64, 65 and 66), in which the minimum values reached by the loss function during optimization grow, instead of decreasing, for a larger value of the parameter controlling the length of the input sequence. Moreover, the metrics for the task of error detection and correction (tables 22, 23, 24 and 25) show a clear drop in performance as the maximum length of the input sequence grows. One of the explanations for this behaviour could be that most of the prepositions used by English learners as a second language are more dependent on an immediate context than on a distant one, which in turn could be explained by the avoidance behaviour of learners. As a consequence, the long sequences not only do not contribute to improving the detection and correction of prepositional errors but start to lose information at some point while processing large sequences.

The classes that seem to be more affected by this parameter are “of” and “that” (figures 67 and 68), meaning that they depend more in particular on short dependencies than the other classes, and if one takes into account that “of” is the class with the larger number of occurrences in the corpus, a fall in performance of the algorithm is expected for large sequences. All other classes keep a similar behaviour for the different values of the sequence length. Thus, the ideal value for the length parameter for the task of detection and correcting prepositional errors is 5.

5.4 Features

Adding features other than lexical ones improve the performance of the algorithm to some extent. On the POS tags side, the use of concatenating an embedding tag vector to the embedding word vector produce better result for the task of detecting and correcting errors (tables 26 - 29) than using only lexical features. However, the performance using POS tags varied depending on the size of the embedding vector, and according to tables 26, 27, 28 and 29, an embedding size of 15 renders the best results for the task at hand – although neither the training graphs (figures 75 - 78) nor the ROC graphs (figures 82 – 84) elucidate any clear difference in the size of the embedding vector. Only the precision-recall graph for the class “on” (figure 80) seems to illustrate a loss in performance for an embedding size of 20, which could mean that large embedding vectors increase the sparsity of the instances of the feature, making them to lose representational power. Furthermore, the use of a threshold to improve the confidence of the algorithm (figures 85 - 88) is a reliable mechanism as it reduces the number of false positives, and a threshold of 95% renders the best results (as has been the case for all experiments so far).

Similarly, the introduction of the index of the parent in the dependency tree showed an improvement over the use of only lexical features. It is illustrated in tables 30, 31, 32 and 33 in which the metrics used in this study show better results for all experiments using the index than the one that did not include this feature. Figures 99 to 102 confirm that the use of a threshold increases the performance of the algorithm, while figures 93 – 98 do not show a different behaviour for the top ten classes other than those seen so far from other experiments. Finally, the optimum size of the embedding vector cannot be identified on the training figures (89 - 92), however, because the metrics reached larger values using small embedding vectors, the final algorithm used an embedding size of 10 is used.

On the other hand, limiting the number of lexical features to the ones occurring more than a given limit in the corpus proved to be a positive limitation to some extent. That is to say, if the number of lexical tokens, occurring say once, are replaced by a generic tag, then the algorithm gained a more powerful representation, which makes sense because sparse features would not provide enough clues whereby the algorithm could learn to distinguish the different classes. However, at the same time, the metrics (tables 34 - 37) show that if the limit starts to be large then the algorithm starts to lose performance because a large number of occurrences would be replaced by a generic tag causing the algorithm to underfit the examples. The use of a threshold proved once again to be an effective way to improve the performance of the algorithm (figures 111 - 115). Finally, although the training figures (103 - 107) do not show a particular difference on varying the lower limit, figures 108 and 109 show that classes such as “in”, “for”, “to” and “from” are better separated for a lower limit of 2, which is confirmed in the metrics for error detection and correction. Therefore, the lower limit chosen in this study is 2.

Moreover, the usage of the original preposition illustrates the most interesting contrasts among the experiments. On the one hand, the introduction of the original preposition produced, during training (figure 119), the lowest value for the loss function during optimization, with the highest accuracy, namely 98% and 99% for validation and training respectively, and the best separating for all classes (figures 121 - 126). However, it produced the worst metrics for the task at hand (although it produced great results for the task of detection when dealing only with the classes that the algorithm targets), and the use of a threshold reduces the algorithm's effectiveness to detect and correct prepositional errors. The high accuracy is caused by the low rate of prepositional errors, i.e. the algorithm learns to identify that for most cases, the use of the original preposition can determine the target class, which is not the case if the goal is to detect prepositional errors as is reflected in the metric tables. Overall, one could say that introducing the original preposition has great potential by combining this with other methods such as populating the corpus with errors depending on statistics and learners' tendencies, but it is outside the scope of this project. On the other hand, replacing the original preposition by a generic tag produced better metrics for the task than the inclusion and exclusion methods (tables 38 - 41), and therefore it was the method used in the final algorithm.

5.5 Embedding

The use of an embedding layer produces better results than introducing the input sequence as a one-hot vector. It is illustrated by figures 130 – 135 in which all classes, with the exceptions being "from" and "to", were better separated when using an embedding input. The training phase (figure 129) shows that the algorithm did not learn to distinguish classes as well as the one using an embedding layer, and additionally it presents a quicker overfitting. Additionally, using the Google ML Engine to train the algorithm, it took about 8 hours when using a one-hot vector, while it takes about 2 hours to train the algorithm using an embedding input (this includes the time taken to create the embedding table). Therefore, the final algorithm uses an embedding layer.

The size of embedding vector for lexical features seems not to have a great impact on the task at hand according to figures 137 and 138, however the metrics in tables 46 – 49 show that for a vector size of 100, the algorithm performs the best. This can be validated looking at figures 140 and 141 in which classes such as "by", "with" and "that" are not well separated when using an embedding size of 50, but in which the algorithm, when using an embedding size of 200, performs worse than an algorithm using an embedding size of 100 for classes such as "in" and "for", which are some of the most popular prepositions and which can potentially contain more errors. Moreover, figures 145 and 146 confirm the positive use of a threshold. Thus, the final algorithm uses an embedding size for lexical features of 100.

Moreover, although the skip-gram model for embedding layers suggests that taking a larger window can provide a more powerful representational vector, the results achieved varying this parameter for the task of error detection and correction show that a smaller window can produce better results than using a larger one (tables 50 - 53). Figures 150 and 151 suggest to some extent that classes such as "to", "of" and "by" are separated in a better way when the window size is 1. Again, this behaviour can be explained by avoidance behaviour showing that for learner data, shorter contexts are more relevant than larger ones, and therefore the final algorithm uses a window size of 1.

5.6 Model

In theory, a bidirectional model is a more powerful representational algorithm than a simple model because the bidirectional model retains a state of two neural networks, one managing a forward sequence and a second one managing a backward sequence. That is to say, in the worst scenario, it should perform as well as a single RNN. However, the results achieved in this study suggest that there is no gain for the task of correcting prepositional errors using a bidirectional model instead of a single RNN (tables 54 - 57). It is also confirmed by analysing the precision-recall graphs as well as the ROC graphs (figures 160 - 165), in which for some classes ("for" and "on") the single model performs better than the bidirectional one, while for other classes ("as", "with", "that" and "from"), it works the other way around. Therefore, because there is no clear advantage in using either a simple model or a bidirectional model, the final algorithm uses a simple model.

One of the components that could be relevant for keeping an informed state of the neural network when dealing with long dependencies was the attention mechanism, but this study has shown that long sequences are not the best alternative when dealing with learner writings. This is illustrated in tables 59 – 62 in which the size of the input sequence was large and the metrics achieved were low in comparison with shorter input lengths. Nonetheless, the metrics show that the algorithm can gain on performance using the attention mechanism, which is illustrated by figures 168 – 173 in which classes such as “of”, “in”, “on“, “with” and “that” are separated in a better way by making use of the attention mechanism. Figure 174 shows that the performance of the attention algorithm is improved by applying a threshold to the decision. Therefore, although the use of the attention mechanism is not relevant for short sequences, the small improvement that it produces was used in the final algorithm.

The algorithms to test the effectiveness of a dropout mechanism were deep recurrent neural networks, i.e. RNN with a relatively large number of layers, namely 8. However, as was seen in the number of layers section, deep neural networks are not a good model for detecting and correcting prepositional errors, with an optimum number of layers of 2. Therefore, the dropout mechanism was not used in the final algorithm. Nonetheless, it is important to highlight that the use of dropout for deep neural networks seems to provide some gain on performances as is illustrated in tables 63 – 66, in which the performance of the algorithms drop as the “keep probability” parameter grows. However, the precision-recall graphs as well as the ROC ones (figures 181 - 186) do not elucidate any clear pattern for the separation of the classes.

Furthermore, the experiments varying the number of classes produced interesting and conclusive results. First, the training figures 193 – 197 show that as the number of classes decreases, the loss function reaches a lower minimum point, i.e. the optimization process works better for a lower number of classes. This behaviour can be expected for the NUCLE corpus, because the number of instances per class in the corpus is quite imbalanced and it decreases sharply for less popular classes (table 4), that is to say, the algorithm can better distinguish the top classes as they have the largest number of instances. Moreover, the precision-recall figures as well as the ROC ones (figures 198 - 213) show that the separation of the classes is not affected to a large extent by the number of classes. For instances, for the top 4 classes the curves are similar for all algorithms. All these findings indicate that if the algorithm is trained using a larger corpus, the performance of the algorithm should be better. The figures also show that for classes with low numbers of occurrences the algorithm does not perform better than just guessing, and that it is not worth targeting classes with very small number of instances. Hence, the final algorithm targets 20 classes.

5.7 Pre-processing

The last experiment before deciding on the final architecture of the algorithm was to test if a context-independent spelling correction was a method worth applying to the data before training and testing the algorithm. The evidence suggests that the use of a spelling corrector improves the performance of the algorithm for the task of prepositional error detection and correction as is illustrated by tables 71 – 74, in which the algorithm performs better when spelling is corrected before introducing the input sequence than when it is not. It can be validated through figures 220 – 225 that show that some classes such as “of”, “from” and “that” are better separated by the algorithm that corrects the spelling of the lexical features. Therefore, the final algorithm uses a spelling correction pre-processing task. It is also important to note that a threshold is also added to the final output as most of the experiments have shown an improvement on performance when the threshold is applied.

5.8 Final Algorithm

The algorithm using the optimum parameters taken from previous experiments, and the final algorithm, which takes the "optimum" algorithm a step further by applying a threshold on decisions for class, were compared with the standard algorithm targeting 20 classes for the tasks of detecting and correcting prepositional errors (tables 76 - 79). The results show that, on the one hand, the optimum algorithm performs better than the standard one, and, on the other hand, the use of thresholds for class makes a considerable improvement in the precision of the algorithm (although reducing the recall) and improves the F1 and F05 score. Moreover, all metrics (tables 76 - 79) present an improvement on the performance of the algorithm by using optimum values, hence one could say that complex models can compensate for the lack of large training data to some extent for the task of detecting and correcting prepositional errors, which also implies that complex algorithms can achieve high quality results if they are trained on large data. These statements can be confirmed by looking at the precision-recall graphs as well as the ROC graphs (figures 228 - 237) in which the optimum algorithm separated all top classes in a better way than the standard one, with a few exceptions for classes such as "at" and "if". However, the optimum algorithm did not show a better performance for the classes presenting few instances in the corpus, meaning that even complex algorithms need a minimum of data to learn to distinguish all classes. Figure 238 confirms the tendency seen so far in which the use of a threshold improves the performance of the algorithm.

Moreover, the comparison with the results achieved by the teams that participated in the CoNLL 2013 shared task show promising results achieved by the final algorithm given that most of the other approaches made use of large external resources to train their algorithm (table 80). It is important to highlight that the number of prepositional errors contained in the testing data provided by CoNLL 2013 is higher in proportion than the number of prepositional errors in the NUCLE corpus. As a consequence, the precision obtained by the algorithm (values in parentheses) is much higher than the values obtained in the testing data generated from NUCLE. The final algorithm seems to perform well even when it only targets prepositional wrong choice errors, meaning that there is room for huge improvement.

6 Conclusions

This research explored the use of recurrent neural networks following a LSTM architecture on the task of detecting and correcting prepositional errors of the wrong choice type. The algorithm is a classifier that aims to predict the correct preposition in the place of the original preposition given a context. The algorithm uses lexical features, POS tags and indexes of the parent in the dependency tree as well as other components that were separately tested to isolate their influence on the final algorithm. It also filters the decisions made by the core algorithm through a threshold-based component which applies a specific threshold for class. The optimum thresholds were obtained from the analysis of the different types of metrics and graphs.

The final results achieved in this study proved that the task of detecting and correcting prepositional errors is a complex one that requires the analysis of several factors to reach appropriate results. Before dealing with prepositional errors, an algorithm must be able to properly identify classes because otherwise the number of false positives that are obtained means the algorithm is not helpful for learners of English as a second language, i.e. if the algorithm marks an error where there is none, the system would confuse a learner rather than help him/her. The low values for the metrics F1 and F05 are due to the low values for the metrics precision, which indicates that the high rate of false positives is an issue for the task of prepositional error detection and correction. Additionally, the use of long sequences of inputs was not an efficient approach for the task at hand as was expected. This was the main reason for using a recurrent neural network that follows an LSTM architecture. The plausible causes are, first, the avoidance behaviour of learners for writing tasks and, second, that most of the prepositions depend on the more immediate context rather than on distant ones (at least for learners).

However, the results achieved in this study are promising if they are compared to the results given by the CoNLL 2013 shared task, meaning that the high rate of false positives is a problem inherent to this task, and not a drawback of the approach followed in this research. Moreover, the possible room for improvement that this study has is huge because all other approaches made use of large corpora, feeding their algorithm with a large number of instances so that their algorithm could learn to identify in a more precise way all prepositional classes, but not obtaining better results than the ones obtained in this study (except for the top three teams which achieved really good results). This hypothesis can be confirmed by the precision-recall figures as well as the ROC graphs which show a better separation for classes containing a larger number of instances, meaning that if the algorithm is trained on a larger corpus the general performance should improve. Furthermore, the relatively good performance of the final algorithm shows both the potential of using neural networks for undertaking NLP tasks and the effectiveness of complex algorithms. However, the different experiments also showed that deep recurrent neural networks are either not suitable for the task of detecting and correcting prepositional errors or need a deeper analysis before building the network, which would require a whole separate study. This also shows that, on the one hand, the literature on techniques to improve recurrent neural networks is huge and a complete study would be required to analyse and test in a deeper way each of the components that were analysed in this study. On the other hand, it proves that machine learning techniques depend too much on data, and that the use of the NUCLE corpus by itself is insufficient to resolve the task at hand, and new approaches using much larger corpora are needed to achieve better results.

The promising results achieved in this study and the room for improvement this approach has, are motives for future work. On the one hand, it would be important to increase the types of errors the algorithm deals with to all errors targeted in the shared tasks CoNLL 2013 and 2014. For most of them the same approach could be used by varying the generation of examples. It is also important to increase the training data as well as the embedding table (it might be by using existing embedding tables trained on huge corpora), and one option might be to use large available native corpora and populate them with learner patterns in order to generate huge learner-like training data. Another experiment that is worth investigating is to find a way to include in an efficient way the original preposition in the training examples, e.g. populating prepositional errors based on the first language of the learner as was carried out by Rozovskaya and Roth. It is also worth testing other approaches on recurrent neural networks such as language models and translators.

In conclusion, the study demonstrates the usability of recurrent neural networks for detecting and correcting prepositional errors made by learners of English as a second language, but at the same time shows that there is still a long way to go before we achieve satisfactory results.

7 Reference List

Abadi, M, Barham, P, Chen, J, Chen, Z, Davis, A, Dean, J, Devin, M, Ghemawat, S, Irving, G & Isard, M 2016, 'TensorFlow: A System for Large-Scale Machine Learning', in *OSDI*, vol. 16, pp. 265-83.

Bahdanau, D, Cho, K & Bengio, Y 2014, 'Neural machine translation by jointly learning to align and translate', *arXiv preprint arXiv:1409.0473*.

Bahl, LR, Jelinek, F & Mercer, RL 1990, 'A maximum likelihood approach to continuous speech recognition', in *Readings in speech recognition*, Elsevier, pp. 308-19.

Baker, J 1975, 'The DRAGON system--An overview', *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 23, no. 1, pp. 24-9.

Baroni, M, Dinu, G & Kruszewski, G 2014, 'Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors', in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, pp. 238-47.

Bengio, Y, Ducharme, R, Vincent, P & Jauvin, C 2003, 'A neural probabilistic language model', *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137-55.

Bird, S, Klein, E & Loper, E 2009, *Natural language processing with Python: analyzing text with the natural language toolkit*, " O'Reilly Media, Inc.".

Bitchener, J, Young, S & Cameron, D 2005, 'The effect of different types of corrective feedback on ESL student writing', *Journal of second language writing*, vol. 14, no. 3, pp. 191-205.

Boyd, A & Meurers, D 2011, 'Data-driven correction of function words in non-native English', in *Proceedings of the 13th European Workshop on Natural Language Generation*, pp. 267-9.

Brants, T & Franz, A 2006, 'Web 1T 5-gram corpus version 1.1', *Google Inc*.

Brockett, C, Dolan, WB & Gamon, M 2006, 'Correcting ESL errors using phrasal SMT techniques', in *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pp. 249-56.

Cambdridge *Prepositions*, viewed 23 of November 2017, <https://dictionary.cambridge.org/grammar/british-grammar/prepositions>.

Chen, D & Manning, C 2014, 'A fast and accurate dependency parser using neural networks', in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 740-50.

Chodorow, M, Dickinson, M, Israel, R & Tetreault, J 2012, 'Problems in evaluating grammatical error detection systems', *Proceedings of COLING 2012*, pp. 611-28.

Chodorow, M, Tetreault, JR & Han, N-R 2007, 'Detection of grammatical errors involving prepositions', in *Proceedings of the fourth ACL-SIGSEM workshop on prepositions*, pp. 25-30.

Chomsky, N 1956, 'Three models for the description of language', *IRE Transactions on information theory*, vol. 2, no. 3, pp. 113-24.

Cohen, J 1960, 'A coefficient of agreement for nominal scales', *Educational and psychological measurement*, vol. 20, no. 1, pp. 37-46.

Collobert, R & Weston, J 2008, 'A unified architecture for natural language processing: Deep neural networks with multitask learning', in *Proceedings of the 25th international conference on Machine learning*, pp. 160-7.

Dagneaux, E, Denness, S & Granger, S 1998, 'Computer-aided error analysis', *System*, vol. 26, no. 2, pp. 163-74.

Dahlmeier, D & Ng, HT 2012, 'Better evaluation for grammatical error correction', in *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 568-72.

Dahlmeier, D, Ng, HT & Ng, EJF 2012, 'NUS at the HOO 2012 Shared Task', in *Proceedings of the Seventh Workshop on Building Educational Applications Using NLP*, pp. 216-24.

Dahlmeier, D, Ng, HT & Wu, SM 2013, 'Building a Large Annotated Corpus of Learner English: The NUS Corpus of Learner English', in *BEA@ NAACL-HLT*, pp. 22-31.

Dale, R, Anisimoff, I & Narroway, G 2012, 'HOO 2012: A report on the preposition and determiner error correction shared task', in *Proceedings of the Seventh Workshop on Building Educational Applications Using NLP*, pp. 54-62.

Dale, R & Kilgarriff, A 2011, 'Helping our own: The HOO 2011 pilot shared task', in *Proceedings of the 13th European Workshop on Natural Language Generation*, pp. 242-9.

Daudaravicius, V, Banchs, RE, Volodina, E & Napoles, C 2016, 'A Report on the Automatic Evaluation of Scientific Writing Shared Task', in *BEA@ NAACL-HLT*, pp. 53-62.

Daumé III, H 2012, 'A course in machine learning', *chapter*, vol. 5, p. 69.

De Felice, R & Pulman, SG 2008, 'A classifier-based approach to preposition and determiner error correction in L2 English', in *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*, pp. 169-76.

Denil, M, Bazzani, L, Larochelle, H & de Freitas, N 2012, 'Learning where to attend with deep architectures for image tracking', *Neural computation*, vol. 24, no. 8, pp. 2151-84.

dos Santos, C & Gatti, M 2014, 'Deep convolutional neural networks for sentiment analysis of short texts', in *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pp. 69-78.

Dredze, M, Crammer, K & Pereira, F 2008, 'Confidence-weighted linear classification', in *Proceedings of the 25th international conference on Machine learning*, pp. 264-71.

El Hihi, S & Bengio, Y 1996, 'Hierarchical recurrent neural networks for long-term dependencies', in *Advances in neural information processing systems*, pp. 493-9.

Elghafari, A, Meurers, D & Wunsch, H 2010, 'Exploring the data-driven prediction of prepositions in English', in *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*, pp. 267-75.

Elman, JL 1990, 'Finding structure in time', *Cognitive science*, vol. 14, no. 2, pp. 179-211.

Felice, M, Yuan, Z, Andersen, ØE, Yannakoudakis, H & Kochmar, E 2014, 'Grammatical error correction using hybrid systems and type filtering', in *CoNLL Shared Task*, pp. 15-24.

Francis, WN & Kucera, H 1964, 'Brown corpus', *Department of Linguistics, Brown University, Providence, Rhode Island*, vol. 1.

Freund, Y & Schapire, RE 1999, 'Large margin classification using the perceptron algorithm', *Machine learning*, vol. 37, no. 3, pp. 277-96.

Fukushima, K, Miyake, S & Ito, T 1983, 'Neocognitron: A neural network model for a mechanism of visual pattern recognition', *IEEE transactions on systems, man, and cybernetics*, no. 5, pp. 826-34.

Gamon, M, Gao, J, Brockett, C, Klementiev, A, Dolan, WB, Belenko, D & Vanderwende, L 2008, 'Using Contextual Speller Techniques and Language Modeling for ESL Error Correction', in *IJCNLP*, vol. 8, pp. 449-56.

Goldberg, Y 2016, 'A Primer on Neural Network Models for Natural Language Processing', *J. Artif. Intell. Res.(JAIR)*, vol. 57, pp. 345-420.

Goodfellow, I, Bengio, Y, Courville, A & Bengio, Y 2016, *Deep learning*, vol. 1, MIT press Cambridge.

Granger, S, Dagneaux, E, Meunier, F & Paquot, M 2009, *International corpus of learner English*, UCL, Presses Univ. de Louvain.

Graves, A, Mohamed, A-r & Hinton, G 2013, 'Speech recognition with deep recurrent neural networks', in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pp. 6645-9.

Gui, S & Yang, H 2003, *Zhongguo Xuexizhe Yingyu Yuliaohu.(Chinese Learner English Corpus). Shanghai Waiyu Jiaoyu Chubanshe*, Chinese.

Gutmann, MU & Hyvärinen, A 2012, 'Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics', *Journal of machine learning research*, vol. 13, no. Feb, pp. 307-61.

Hakuta, K 2011, 'Educating language minority students and affirming their equal rights: Research and practical perspectives', *Educational Researcher*, vol. 40, no. 4, pp. 163-74.

Han, N-R, Chodorow, M & Leacock, C 2006, 'Detecting errors in English article usage by non-native speakers', *Natural Language Engineering*, vol. 12, no. 2, pp. 115-29.

Hermans, M & Schrauwen, B 2013, 'Training and analysing deep recurrent neural networks', in *Advances in neural information processing systems*, pp. 190-8.

Hochreiter, S & Schmidhuber, J 1997, 'Long short-term memory', *Neural computation*, vol. 9, no. 8, pp. 1735-80.

Hodgkin, AL & Huxley, AF 1952, 'The dual effect of membrane potential on sodium conductance in the giant axon of Loligo', *The Journal of physiology*, vol. 116, no. 4, pp. 497-506.

Hornik, K, Stinchcombe, M & White, H 1989, 'Multilayer feedforward networks are universal approximators', *Neural networks*, vol. 2, no. 5, pp. 359-66.

Huddleston, R & Pullum, GK 2005, *A student's introduction to English grammar*, Cambridge University Press.

Irsoy, O & Cardie, C 2014, 'Opinion mining with deep recurrent neural networks', in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 720-8.

Izhikevich, EM 2003, 'Simple model of spiking neurons', *IEEE Transactions on neural networks*, vol. 14, no. 6, pp. 1569-72.

Izumi, E, Uchimoto, K & Isahara, H 2004, 'SST speech corpus of Japanese learners' English and automatic detection of learners' errors', *ICAME Journal*, vol. 28, pp. 31-48.

Izumi, E, Uchimoto, K, Saiga, T, Supnithi, T & Isahara, H 2003, 'Automatic error detection in the Japanese learners' English spoken data', in *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 2*, pp. 145-8.

James, C 2013, *Errors in language learning and use: Exploring error analysis*, Routledge.

Jaynes, ET 1957, 'Information theory and statistical mechanics', *Physical review*, vol. 106, no. 4, p. 620.

Jelinek, F, Bahl, L & Mercer, R 1975, 'Design of a linguistic statistical decoder for the recognition of continuous speech', *IEEE Transactions on Information Theory*, vol. 21, no. 3, pp. 250-6.

Jozefowicz, R, Vinyals, O, Schuster, M, Shazeer, N & Wu, Y 2016, 'Exploring the limits of language modeling', *arXiv preprint arXiv:1602.02410*.

Jurafsky, D & Martin, JH 2009, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, MIT Press.

Katz, S 1987, 'Estimation of probabilities from sparse data for the language model component of a speech recognizer', *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 35, no. 3, pp. 400-1.

Kingma, DP & Ba, J 2014, 'Adam: A method for stochastic optimization', *arXiv preprint arXiv:1412.6980*.

Klein, D & Manning, CD 2003, 'Accurate unlexicalized parsing', in *Proceedings of the 41st annual meeting of the association for computational linguistics*.

Koehn, P, Och, FJ & Marcu, D 2003, 'Statistical phrase-based translation', in *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pp. 48-54.

Landis, JR & Koch, GG 1977, 'The measurement of observer agreement for categorical data', *biometrics*, pp. 159-74.

Larochelle, H & Hinton, GE 2010, 'Learning to combine foveal glimpses with a third-order Boltzmann machine', in *Advances in neural information processing systems*, pp. 1243-51.

Leacock, C, Chodorow, M, Gamon, M & Tetreault, J 2014, 'Automated grammatical error detection for language learners', *Synthesis lectures on human language technologies*, vol. 7, no. 1, pp. 1-170.

Lee, J & Seneff, S 2006, 'Automatic grammar correction for second-language learners', in *Ninth International Conference on Spoken Language Processing*.

Lindstromberg, S 2010, *English prepositions explained*, John Benjamins Publishing.

Liu, Z-R & Liu, Y 2017, 'Exploiting Unlabeled Data for Neural Grammatical Error Detection', *Journal of Computer Science and Technology*, vol. 32, no. 4, pp. 758-67.

Manning, C 2005, 'Maxent models and discriminative estimation', *CS 224N lecture notes, Spring*.

Manning, CD, Raghavan, P & Schütze, H 2008, 'Text classification and naive bayes', *Introduction to information retrieval*, vol. 1, p. 6.

Marcus, MP, Marcinkiewicz, MA & Santorini, B 1993, 'Building a large annotated corpus of English: The Penn Treebank', *Computational linguistics*, vol. 19, no. 2, pp. 313-30.

Markov, AA 1913, 'Calculation of probabilities', *St Petersburg*.

McCulloch, WS & Pitts, W 1943, 'A logical calculus of the ideas immanent in nervous activity', *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115-33.

McGregor, RJ 1987, 'Neural and brain modelling', *Academic Press, San Diego*, vol. 112, pp. 245-58.

Mikolov, T, Chen, K, Corrado, G & Dean, J 2013, 'Efficient estimation of word representations in vector space', *arXiv preprint arXiv:1301.3781*.

Mikolov, T, Karafiát, M, Burget, L, Černocký, J & Khudanpur, S 2010, 'Recurrent neural network based language model', in *Eleventh Annual Conference of the International Speech Communication Association*.

Mikolov, T, Sutskever, I, Chen, K, Corrado, GS & Dean, J 2013, 'Distributed representations of words and phrases and their compositionality', in *Advances in neural information processing systems*, pp. 3111-9.

Minsky, M & Papert, SA 1969, *Perceptrons: An introduction to computational geometry*, MIT press.

Mizumoto, T, Hayashibe, Y, Komachi, M, Nagata, M & Matsumoto, Y 2012, 'The effect of learner corpus size in grammatical error correction of ESL writings', *Proceedings of COLING 2012: Posters*, pp. 863-72.

Morin, F & Bengio, Y 2005, 'Hierarchical Probabilistic Neural Network Language Model', in *Aistats*, vol. 5, pp. 246-52.

MoveHub 2018, *Second Languages Around the World*, viewed January 2nd 2018, <https://www.movehub.com/blog/global-second-languages/>.

Ng, HT, Wu, SM, Briscoe, T, Hadiwinoto, C, Susanto, RH & Bryant, C 2014, 'The CoNLL-2014 Shared Task on Grammatical Error Correction', in *CoNLL Shared Task*, pp. 1-14.

Ng, HT, Wu, SM, Wu, Y, Hadiwinoto, C & Tetreault, J 2013, 'The CoNLL-2013 Shared Task on Grammatical Error Correction', in *CoNLL Shared Task*.

Nicholls, D 2003, 'The Cambridge Learner Corpus: Error coding and analysis for lexicography and ELT', in *Proceedings of the Corpus Linguistics 2003 conference*, vol. 16, pp. 572-81.

Och, FJ & Ney, H 2003, 'A systematic comparison of various statistical alignment models', *Computational linguistics*, vol. 29, no. 1, pp. 19-51.

Olah, C 2015, 'Understanding lstm networks, 2015', *URL https://colah. github. io/posts/2015-08-Understanding-LSTMs*.

Pascanu, R, Mikolov, T & Bengio, Y 2013, 'On the difficulty of training recurrent neural networks', in *International Conference on Machine Learning*, pp. 1310-8.

Pei, W, Ge, T & Chang, B 2015, 'An effective neural network model for graph-based dependency parsing', in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, vol. 1, pp. 313-22.

Pham, V, Bluche, T, Kermorvant, C & Louradour, J 2014, 'Dropout improves recurrent neural networks for handwriting recognition', in *Frontiers in Handwriting Recognition (ICFHR), 2014 14th International Conference on*, pp. 285-90.

Powers, DM 2012, 'The problem with kappa', in *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 345-55.

Python 2018, *Python*, viewed January 2 2018, <https://www.python.org/>.

Rosenblatt, F 1958, 'The perceptron: a probabilistic model for information storage and organization in the brain', *Psychological review*, vol. 65, no. 6, p. 386.

Rozovskaya, A, Chang, K-W, Sammons, M & Roth, D 2013, 'The University of Illinois system in the CoNLL-2013 shared task', in *Proceedings of the Seventeenth Conference on Computational Natural Language Learning: Shared Task*, pp. 13-9.

Rozovskaya, A & Roth, D 2010a, 'Annotating ESL errors: Challenges and rewards', in *Proceedings of the NAACL HLT 2010 fifth workshop on innovative use of NLP for building educational applications*, pp. 28-36.

—— 2010b, 'Generating confusion sets for context-sensitive error correction', in *Proceedings of the 2010 conference on empirical methods in natural language processing*, pp. 961-70.

—— 2010c, 'Training paradigms for correcting errors in grammar and usage', in *Human language technologies: The 2010 annual conference of the north american chapter of the association for computational linguistics*, pp. 154-62.

—— 2011, 'Algorithm selection and model adaptation for ESL correction tasks', in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pp. 924-33.

Rozovskaya, A, Sammons, M, Gioja, J & Roth, D 2011, 'University of Illinois system in HOO text correction shared task', in *Proceedings of the 13th European Workshop on Natural Language Generation*, pp. 263-6.

Rozovskaya, A, Sammons, M & Roth, D 2012, 'The UI system in the HOO 2012 shared task on error correction', in *Proceedings of the Seventh Workshop on Building Educational Applications Using NLP*, pp. 272-80.

Schuster, M & Paliwal, KK 1997, 'Bidirectional recurrent neural networks', *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673-81.

Shannon, CE 1948, 'A mathematical theory of communication, Part I, Part II', *Bell Syst. Tech. J.*, vol. 27, pp. 623-56.

Siganos, D & Stergiou, C 1996, 'Neural networks, the human brain, and learning', *Imperial College London: Surveys and Presentations in Information Systems Engineering*.

Srivastava, N 2013, 'Improving neural networks with dropout', *University of Toronto*, vol. 182.

Steiner, B 2015, *Tensorflow word2vec tutorial*, viewed 09 October 2017, <https://github.com/tensorflow/tensorflow/blob/r1.5/tensorflow/examples/tutorials/word2vec/word2vec_basic.py>.

Sutskever, I, Vinyals, O & Le, QV 2014, 'Sequence to sequence learning with neural networks', in *Advances in neural information processing systems*, pp. 3104-12.

Tetreault, JR & Chodorow, M 2008, 'The ups and downs of preposition error detection in ESL writing', in *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*, pp. 865-72.

Training, DoEa 2017, *International Student Data 2017*, viewed January 2sd 2018, <https://internationaleducation.gov.au/research/International-Student-Data/Pages/InternationalStudentData2017.aspx>.

Vasilis, V 2015, *Machine Learning Tutorial: The Naive Bayes Text Classifier*, <http://blog.datumbox.com/machine-learning-tutorial-the-naive-bayes-text-classifier/>.

Vaswani, A, Zhao, Y, Fossum, V & Chiang, D 2013, 'Decoding with large-scale neural language models improves translation', in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1387-92.

Vinyals, O, Kaiser, Ł, Koo, T, Petrov, S, Sutskever, I & Hinton, G 2015, 'Grammar as a foreign language', in *Advances in Neural Information Processing Systems*, pp. 2773-81.

Weiss, D, Alberti, C, Collins, M & Petrov, S 2015, 'Structured training for neural network transition-based parsing', *arXiv preprint arXiv:1506.06158*.

Werbos, P 1974, 'Beyond regression: new fools for prediction and analysis in the behavioral sciences', *PhD thesis, Harvard University*.

Werbos, PJ 1990, 'Backpropagation through time: what it does and how to do it', *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550-60.

Xu, W, Auli, M & Clark, S 2015, 'CCG supertagging with a recurrent neural network', in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, vol. 2, pp. 250-5.

Yannakoudakis, H, Briscoe, T & Medlock, B 2011, 'A new dataset and method for automatically grading ESOL texts', in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pp. 180-9.

Yoshimoto, I, Kose, T, Mitsuzawa, K, Sakaguchi, K, Mizumoto, T, Hayashibe, Y, Komachi, M & Matsumoto, Y 2013, 'NAIST at 2013 CoNLL Grammatical Error Correction Shared Task', in *CoNLL Shared Task*, pp. 26-33.

Zaremba, W, Sutskever, I & Vinyals, O 2014, 'Recurrent neural network regularization', *arXiv preprint arXiv:1409.2329*.

Zeng, D, Liu, K, Lai, S, Zhou, G & Zhao, J 2014, 'Relation classification via convolutional deep neural network', in *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pp. 2335-44.