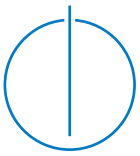
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Semi-supervised Learning for Biomedical Named-Entity Recognition

Aleksandar Bojchevski



DEPARTMENT OF INFORMATICS

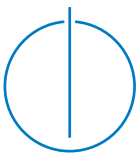
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Semi-supervised Learning for Biomedical Named-Entity Recognition

Semi-überwachtes Lernen für Namenserkennung

Author:	Aleksandar Bojchevski
Supervisor:	Prof. Dr. Burkhard Rost
Advisor:	Juan Miguel Cejuela
Submission Date:	December 15, 2015



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, December 15, 2015 Aleksandar Bojchevski

Acknowledgments

This master's thesis was developed as a joint project at Rostlab, the Bioinformatics chair at Technische Universität München, in conjunction with the bachelor's thesis of Carsten Uhlig. The project and both theses were under the supervision of Prof. Dr. Burkhard Rost and Juan Miguel Cejuela as the main advisor. Many thanks to the whole Rostlab team for the support and opportunities they offered me.

I would like to express immense gratitude to my advisor Juan Miguel Cejuela for his continued support during not only this thesis, but the better half of my master studies as well.

Many thanks to Carsten Uhlig, whom I consider a true scholar and friend. The final nala method including much of the implementation was a joint effort by both of us. With his related work on relationship extraction, Ashish Baghudana contributed as well to the development of the general nalaf framework.

Special thanks to DAAD (Deutscher Akademischer Austauschdienst) for granting me a scholarship during the full duration of my master studies, thus enabling my academic development in Germany.

Last but not least, I am grateful to my family for their unconditional support and understanding.

Abstract

As the volume of published research in the biomedical domain increases, the need for effective information extraction systems grows in parallel. In this context, the task of named-entity recognition (NER) is essential. NER is defined as the classification of words in free text that represent predefined categories such as genes, proteins or other entities.

As a specific application of NER, the main focus of this thesis is the recognition of mutation mentions from the biomedical literature. More specifically we aim to create a model able to recognize mutation mentions expressed in natural language. The current state-of-the-art method, tmVar [1] is only able to recognize a small subset of standard or semi-standard mentions. Our method both outperforms tmVar on those types of mentions and is also able to recognize natural language (NL) mentions. Previously no other method considered NL mutation mentions.

The performance of NER machine learning models is intrinsically limited by the availability of high-quality-annotated corpora. The construction of such corpora is costly – specially when expert annotators are required. In the biomedical domain, the difficulty of the task is even greater, since the number of possible named entities is higher and keeps growing with new discoveries.

To combat the lack of large annotated corpora, we turn to the exploitation of large volumes of unlabeled text, applying a semi-supervised learning approach. Using techniques for unsupervised feature learning we aim to increase the performance of traditional NER models. More specifically, this thesis focuses on augmenting common conditional random field (CRF) approaches combined with novel word representation features learned from large bodies of biomedical text. Furthermore, using an active learning approach we extend an existing corpus of mutation mentions (IDP4 [2]) with additional NL mentions. Finally, and in support of evaluating our semi-supervised learning approach, we develop a complete pipeline for biomedical named-entity recognition including preprocessing steps, feature generation, model learning and normalized predictions. Our extended corpus, NER tool and pipeline framework are all open sourced on GitHub (`https://github.com/Rostlab/nala`).

Contents

1 Introduction

The topic of this thesis in general is semi-supervised learning for biomedical named-entity recognition. More specifically, we focus on learning entities representing mutation mentions, with emphasis on mentions expressed in natural language. The rest of the introduction to this thesis is organized in three parts. First, we state our goals that we have set for ourselves at the beginning of the project. Next, we give a brief overview of biomedical natural language processing in general, including motivations and significance. Finally, we give an overview of the landscape of existing and state-of-the-art methods for mutation mention extraction.

1.1 Goals

In this section, we state the goals for this thesis and the method in general, as defined at the beginning of the project. All of our research efforts were aimed at achieving these goals and answering the questions we posed as reliably and rigorously as possible. The goals are as follows:

1. Study the significance of natural language (NL) mentions in mutation mention recognition
2. Create a corpus (or extend an existing one) for NL mentions, if the study in the first goal proves them to be significant enough
3. Crete a method for mutation mention extraction satisfying the following constraints:
    - The method should perform as good as, or better than the current state-of-the-art method tmVar [1].
    - The method should perform fairly well for NL mentions, should they prove to be significant.
    - Create a tool based on our method, that can be easily adopted and extended by the community. This entails writing well-documented, extensible and modular code as well as a good end-user documentation and a simple API easily usable by both computer scientists and biomedical researchers.

1.2 Biomedical NLP

Biomedical text mining or natural language processing (NLP), is a relatively recent research area focusing on applying text mining techniques to biomedical literature. Most of the efforts in the field are focused towards automatic information extraction, such as identification of proteins, genes, mutations, diseases or chemical compounds as well as the relations between them. More generally, we can say the goal of biomedical NLP systems is efficient knowledge discovery, and ultimately accelerating new biomedical discoveries.

As the volume of published research in the biomedical domain increases, exhibiting exponential growth [3], the need for effective information extraction systems grows with it. Luckily, the ratio of articles available for text mining, stored in online databases (e.g. MEDLINE/PubMed) is growing as well. However, directly adopting methods that are proven to work well in other areas, for example named entity recognition in news articles, is not straightforward. This in big part is due to the unstandardized, constantly changing terminology and complex nomenclature in the biomedical field. Coupled with the unstructured nature of natural language, this makes any BioNLP task challenging. Furthermore, there is a lack of structural representation of domain knowledge and limited availability of quality annotated corpora for model training.

The major challenge as seen by many [4, 5, 6] is to create tools that are truly useful to the biomedical community, helping them solve real world problems. Therefore, close collaboration between computer scientists, biomedical researchers and article publishers is required. Efforts have to be put into making more published literature available for text mining purposes, moving away from processing only abstracts and towards processing full text articles. And finally, we need to develop models specifically tailored to BioNLP.

1.3 Mutation Mention Extraction: Landscape of Methods

MutationFinder [7] is one of the oldest and still rather widely used system for extraction of point mutation mentions. It is based on set of approximately 700 regular expressions and has good precision (98.4%) and satisfactory recall (81.9%). The main disadvantage of this method is that it can only recognize a small subset of simple standard mentions.

Another system is OSIRISv1.2 [8], it focuses on single nucleotide polymorphisms (SNPs) and is based on a pattern-based search algorithm. Performs both extraction of variation terms from articles and mapping to their respective dbSNP identifiers. With 99% precision and 82% recall, it performs fairly well, but similarly to MutationFinder it is quite limited in that it is only able to predict one type of simple standard mentions.

Another system for NER of genetic variants, that similarly focus on SNPs is SETH [9]. This system implements an Extended Backus–Naur (EBNF) grammar, to recognize mentions as defined by the Human Genome Variation Society (HGVS) nomenclature [10]. Additionally, it modifies the original implementation of MutationFinder and extends it with new regular expressions aimed at recognizing mentions that do not strictly follow the HGVS nomenclature. Furthermore, it performs grounding and normalization to dbSNP identifiers. Similarly as all of the above mentioned systems, SETH achieves good performance (precision 0.98%, recall 0.86%), but it is limited to simple standard mentions.

1.3.1 State-of-the-art Method

Currently, the state-of-the-art method for mutation mention extraction as considered by many is tmVar [1]. This method is based on conditional random fields (CRFs) and can capture significantly more diverse types of mentions compared to the previous methods. The success of this method is mostly due to the following key characteristics:

- Use of a CRF based machine learning approach, previously not used for mutation mention extraction, as opposed to regular expressions and rule based approaches.
- Combination of typical NER features with a wide array of hand crafted features specific to mutation mentions.
- Custom defined fine grained tokenization that attempts to separate different components of a mutation mention into separate tokens (e.g. token for the part signifying position, wild-type, frame-shift, etc.), as well a complimentary set of 11 fine grained labels as opposed to the typical labels using the BIO or the BIEO format.
- A set of additional postprocessing steps, attempting to correct misclassified mentions.

All of these characteristics amount to a method that yields great performance with precision and recall of around 91%.

However, although it does recognize more varied types of mentions compared to previous methods, it is still not able to handle mentions expressed in natural language. For a definition on natural language mentions and analysis of why it is important for a method to recognize such mentions please refer to chapter 2.

2 Natural Language Mutation Mentions

All of the methods that we discussed in section 1.3, perform relatively well on a subset of standard mutation mentions. However, most of them are not able to handle mentions expressed in natural language. As stated previously, one of the goals of this thesis was to study the significance of NL mentions and develop a model able to recognize them should they prove to be significant. In order to do so, first we have to formulate a working definition of what constitutes an natural language mention.

2.1 Definition

Developing a reliable definition of what makes a mutation mention natural language (NL) or standard (ST) is not a straightforward task. Even human annotators may disagree whether a particular mention is NL or not. For example the mention *'alanine 27 substitution for valine'* maybe be considered by some to be an NL mention, since it is not expressed with the standard HGVS nomenclature[10]. Others however, may consider the same mention to be standard or semi-standard, since one can easily write simple regular expressions to match such mentions. Therefore, we wanted to algorithmically define NL mention, using simple heuristics, easing our further analysis.

We implemented this task as a preprocessing step in our pipeline, where different NL definitions implement the interface `nala.preprocessing.definers.NLDefiner`. To accommodate for the gray area between completely standard and natural language mentions, our definers can define a mention as one of 3 subclasses detailed in Table 2.1, while examples for each of them is given in Table 2.2.

We implemented several different definitions. Note that most of the implementation of NL definers was done by Carsten Uhlig. They are:

mention type	description	method
(ST)	HGVS nomenclature; regular patterns	MutationFinder [7]
semi-standard (SS)	twilight zone, simple NL words	tmVar [1]
natural language (NL)	expressed in natural language; anything not ST or SS	nala

Table 2.1: Description of subclasses of mutation mentions and examples of methods tailored at recognizing them.

subclass	example
ST	c.187C>A
SS	Gln 187 to Lys
NL	substitution of a single amino acid (Gln to Lys), which is caused by a point mutation of a single nucleotide (C to A) in exon 3 at position 187

Table 2.2: Examples of typical mutation mentions for each subclass.

- **InclusiveNLDefiner** is based on two simple heuristics: length of the mention and number of words (here by words we mean the string split by empty spaces). If the mention has length greater than a set minimum length (by default 18) and more words than the minimum number of words (by default 4) then we consider it an NL mention, otherwise an ST mention. This definer cannot distinguish the SS subclass. We call it inclusive since we explicitly define which mentions are NL mentions.
- **SimpleExclusiveNLDefiner** uses different sets of regular expressions in conjunction with the number of words heuristic. The regular expressions try to match ST mentions, since they are easier to formulate as patterns. Anything not matched by them is considered an NL mention, hence the name exclusive. This definer cannot distinguish the SS subclass.
- **ExclusiveNLDefiner** is an extension of SimpleExclusiveNLDefiner with additional regular expressions aimed at capturing the SS class.
- **TmVarRegexNLDefiner** is another exclusive type of definer, in this case we used regular expressions that try to match ST mentions, which are taken from the tmVar system. Anything not matched by them is considered an NL mention. This definer cannot distinguish the SS subclass.
- **TmVarNLDefiner** is another exclusive approach, where we use the complete tmVar system instead of just their regular expressions. If tmVar is able to predict a mention then we consider it an ST mention otherwise we consider it a NL mention. This definer cannot distinguish the SS subclass, however, since tmVar can somewhat capture SS mentions, in this case it means that SS mentions are categorized as part of the ST subclass.

2.1.1 Manual Definition

The algorithmic definers implemented were intended to be used during the development of the method as a guiding tool to explore the performance on different types

of mentions. The idea is to run some definer on our dataset after we collected the predictions and see the effect of our changes reflected in each subclass separately. However, they are ultimately based on simple heuristics and for the task of determining NL significance we wanted a more reliable source. Therefore, we manually annotated each of the mentions in the IDP4 corpus [2] for 2 separate annotators and additionally in the Verspoor courpus [11]. For each article we recorded:

- Total number of mentions.
- Number of NL mentions.
- Number of NL mentions that do not exist as a standard mention within the same text.

2.2 Significance

Now that we have working definitions of what is an NL mention we are ready to answer the question if they are significant.

First, in order to explore the difference between abstracts and full texts we define:

$$RatioAbstractFull = \frac{NLAbstract}{NLFull}$$
$$where\ NLAbstract = \frac{NLmentionsAbstract}{TokensAbstract}$$
$$and\ NLFull = \frac{NLmentionsFull}{TokensFull}$$

We normalize the NL mutation mentions by the number of tokens to better represent the amount of text one person would have to read through manually on average to find a NL mention. After calculating the ratios we concluded that we prefer abstracts because the ratio is higher than 1, with namely IDP4 corpus having 5 times more NL mentions in the abstracts and Verspoor containing around 1.8 times more NL mentions in the abstracts.

Next, we wanted to examine the prevalence of NL mentions which should be a good indicator of their significance. More specifically, we look at the fraction of NL mentions from all mentions as defined by one of the NL definers. The results vary for different corpora.

The tmVar corpus has only 3% NL mentions, this makes sense, since the tmVar system was not aimed at NL mentions at all. In the Verspoor corpus, there are 10% to 30% NL mentions depending on the used NLDefiner. However, after manual inspection of those mentions we concluded they are too broad (e.g. mutation in exon 5), and

do not follow our annotation guidelines defined for the IDP4 corpus. Therefore, we excluded them from any training or further analysis. The SETH corpus focuses on ST mentions like tmVar, and similarly has less than 5% of NL mentions. Finally, the IDP4 corpus has 10% of NL mentions, which we deem significant enough to continue with further analysis.

In conclusion, only the IDP4 corpus has a significant amount of usable NL mentions, while the other corpora have mostly ST or SS mentions. Note that most of the analysis on abstract vs. fulls text and the prevalence of nl mentions was done by Carsten Uhlig.

The second analysis in regards to the NL significance, focuses on finding the percentage of mutation mentions in natural language that do not appear as standard mention. In other words if some mention appears as both NL and standard mention within the same text, then the significance is significantly lower compared to mentions of mutations, that only appear as NL mention and are not translated into a standard mention.

Using the 3 numbers we recored for the manual definition described in subsection 2.1.1 we calculate the following statistics:

- **S1**: percentage of documents that have at least one NL mention.
- **S2**: percentage of documents that have NL mention and at least one non-ST mention.
- **S3**: percentage of documents that have at least one non-ST mention.
- **S4**: average percentage of non-ST mentions per document.
- **S5**: ratio of NL mentions from ALL mentions.
- **S6**: ratio of non-ST mentions from ALL mentions.

The results of those statistics are present in Table 2.3. The main three insights we can gain from these results are the following:

- Between **32%** and **45%** (depending on corpus) of documents contain at least one mention that is an NL mention and is not translated into an ST mention within the same text.
- Between **5%** and **12%** (depending on corpus) of total mentions are NL mentions that are not translated into an ST mention within the same text.
- When considering abstracts vs. full texts, the ratio of NL mentions is significantly different. More specifically, around **8%** for full texts, compared to between **14%** and **19%** for abstracts. With this we confirm the previous conclusion based

	IDP4: Annotator 1			IDP4: Annotator 2			Verspoor
statistic	A+F	A	F	A+F	A	F	F
S1	61.96%	61.67%	62.50%	69.15%	67.19%	73.33%	100.00%
S2	50.88%	48.65%	55.00%	64.62%	62.79%	68.18%	40.00%
S3	**31.52%**	30.00%	34.38%	**44.68%**	42.19%	50.00%	**40.00%**
S4	36.28%	41.35%	26.90%	53.31%	53.80%	52.36%	3.84%
S5	24.31%	27.14%	22.49%	21.33%	30.03%	16.12%	73.87%
S6	**10.19%**	14.13%	7.66%	**12.01%**	18.77%	7.96%	**4.95%**

Table 2.3: Values for the $S1 - S6$ statistics shown for two randomly selected annotators from the IDP4 corpus and the Verspoor corpus. A+F signifies calculation for both abstracts and full texts together, while A or F signifies abstract or full text on their own respectively.

on algorithmic definers, that abstracts indeed are more useful for finding NL mentions.

These three insights, along with the first analysis, clearly demonstrate that NL mentions are significant enough. Therefore, our goal of creating a model that can reliably predict them is justified.

3 Materials and Methods

3.1 Preliminaries

3.1.1 Corpora

The creation of reliable NER models, in more ways than one, is highly dependent on the quality of the annotated corpora used for training and evaluation. Unfortunately, in the biomedical domain, not a lot of corpora exist regarding mutation mentions. Additionally, most of the existing corpora have a relatively small number of documents, and all of them with the exception of the IDP4 corpus do not consider NL mentions at all. This section briefly describes the corpora that were used during the development process of our method. We evaluated the following copora:

- The **IDP4** corpus [2] is a gold standard corpus, that compared to related corpora, that only have annotations of standard mutation mentions, additionally contains annotations of natural language mentions of mutations, annotations of proteins and organisms, as well as relations between them. The quality of the corpus is reflected by the high inter-annotator agreement (IAA), reported at 87.23% for all documents and ranging from around 85% to around 90% between pairs of annotators. The corpus also defines a comprehensive list of annotation rules, available at `https://www.tagtog.net/jmcejuela/IDP4/-settings`, which were originally used to create it and can be used now to extend it. This corpus was developed as part of previous work, in the frame of an interdisciplinary project in the biomedical area, and is the main corpus on which we build our method. The corpus consists of 163 documents, 85 of which are abstracts and 78 are full text documents.

- **tmVar** corpus is provided from the state-of-the-art method tmVar [1] and was mostly used as a benchmarking corpus. It contains 500 abstract with mostly standard mentions and few semi-standard mentions. As stated previously in our goals, we want to reproduce tmVar as baseline for our framework, therefore we use it to evaluate the performance and meaningfully compare tmVar to nala.

- The **Verspoor** corpus [11], also known as the Variome Corpus, is a relatively small corpus of 10 full text documents. Besides mutation mentions it has other

annotations as well such as: body parts, diseases, genes, etc. As discussed in section 2.2, although it has many NL mentions, most of them are not suitable according to the IDP4 annotation guidelines. Therefore, this corpus was not included for developing our method.

- The **MutationFinder** corpus is a gold standard corpus for mutation extraction systems consisting of 588 abstracts used to develop the MutationFinder method [7]. This corpus has only simple standard mentions.
- The **SETH** corpus contains 630 abstracts used as the main corpus for the SETH method, and similarly has only simple standard mentions [9].

3.1.2 Merging Strategies

In the IDP4 corpus, 4 annotators in total were annotating documents, with each one annotating around 100 abstract and full-text articles. The distribution of the documents to different annotators was varied. Some documents were annotated by all 4 annotators, while others were annotated by groups of 2 annotators. Some documents, after the annotation rules were already developed and the annotators had enough experience were annotated only once.

In the current state of the corpus, the annotations are provided separately per each annotator. In the early stages of development we only used the annotations from one randomly selected annotator. However, to fully utilized the corpus the annotations have to merged together for those documents that have been annotated more than once. In order to achieve that we developed the following merging scheme:

1. Start with the annotations from the first annotators as base
2. Merge the annotation from each subsequent annotator into the base, one at a time, using one of the following strategies:
   a) For the general merging strategy choose between: **union** or **intersection**
      - where union merges all entities, overlapping and non-overlapping between annotators, handling cases of partial overlap according to 2.b.
      - and intersection merges only overlapping entities and discards the non-overlapping ones, handling cases of partial overlap according to 2.b.
   b) For the partially overlapping entity merging strategy choose between: **shortest** or **longest** entity or **priority**
      - where shortest picks the entity with shorter length out of the two.
      - longest picks the entity with longer length out of the two.

- and priority picks the entity from the annotator with higher priority, where the priority of each annotator was assigned according the their average inter annotator agreement (IAA), with higher average IAA translating to higher priority.

3.2 Evaluation Measures

In supervised learning scenarios, to estimate the performance of a method, often the measures used are based on simple counting of the true positives, false positives, true negatives and false negatives. With those four counts measures such as accuracy, precision, recall and F_1 score are calculated. In typical classification tasks, we are considering the labels of each instance separately. If we translate this directly to our task, it means to report the performance based on the misclassification of individual tokens. However, such evaluation is not suitable due to several reasons. First, it will greatly overestimate the real performance, since most of the individual token labels are *'O'* which are easy to predict. Second, viewed from a point of a higher level of abstraction, e.g. from the end-user's perspective, the tokens do not have any meaning. What we care about is whether a span of text is a mutation mention or not, and not if we correctly predicted each individual token's label in that span.

Therefore, we use measures defined on a mention level. At this level, we count a prediction as a true positive, if and only if, it matches exactly the same offset as the real mention, or in other words if they span the same tokens. If a prediction does not match exactly to a real mention we have two cases to consider. The prediction and the real mention:

- either do not overlap at all, in which case we have a false positive or a false negative accordingly.
- or they overlap partially. What we do in this case is defined by our strictness parameter. If the **strictness** is set to '**exact**' we count this a misclassification since for this strictness only mentions with the exact same span are considered true positives. However, if the strictness is set to '**overlapping**' then we do count it as true positive. A third setting of strictness was used as well, termed '**half-overlapping**', where we divide the number of partial matches by two.

Using these mention level counts we calculate:

$$Precision = \frac{tp}{tp + fp}$$
$$Recall = \frac{tp}{tp + fn}$$
$$F_1\ Score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Another evaluation measure that makes sense is a document level evaluation measure. However, due to time restrictions we were not able to implement it and will be implemented in future releases of nalaf.

3.3 System Architecture

In order to facilitate the accomplishment of the goals we stated in the introduction we created a pipeline for biomedical named-entity recognition, including preprocessing, feature generation, model learning and other modules. The pipeline was implemented in Python, supporting versions 3.2 or greater. Python was an obvious choice due to its wide adoption by the general scientific community. Furthermore, Python allows for easy prototyping avoiding excessive boilerplate code.

On figure Figure 3.1 one can see an overview of the system architecture.

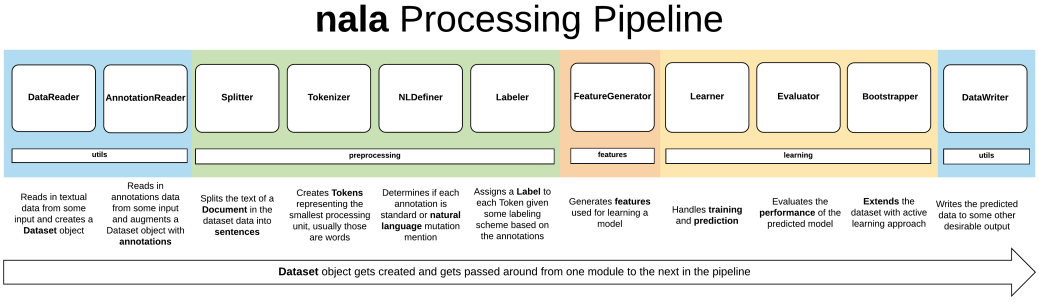


Figure 3.1: nala system architecture

3.3.1 Design Goals

From the very start the key design goal was to write code that is highly modular, with minimal dependencies between modules and well defined interfaces. The motivations

behind this design goal were several. Firstly, it allows for creation of software that can be easily used and extended by the community. This is important, since our system will be opened sourced. Secondly, it allows us to easily experiment and change different components of the learning process, in hope to improve the performance. For example, one can easily swap out one implementation of a `Tokenizer` for another, without having to worry about the rest of modules. Similarity, one can easily extend the system with new features, by adding a new implementation of a `FeatureGenerator`, or create a new combination from existing ones. Furthermore, we wanted to make sure that the code is well documented and well tested. Therefore, a set of unit tests have been implemented as well for each of the key modules in our pipeline.

Initially, the implementation was self contained as a single project. However, after realizing that a big portion of the already implemented functionality is not necessarily specific to biomedical NER, we decided to split it into two separate projects, nalaf and nala.

3.3.2 nalaf

The goal of nalaf - (Na)tural (La)nguage (F)ramework - is to be a general-purpose module-based and easy-to-use framework for common text mining tasks. At the moment two tasks are covered: named-entity recognition (NER) and relationship extraction. These modules support both training and annotating. Associated to these, helper components such as cross-validation training or reading and conversion from different corpora formats are given.

We created nalaf by extracting all of the code from the original project that was not tied to the biomedical domain, thereby enabling users to train and use NER models from any domain. For example, modules for generating typical NER features such as word shapes or part-of-speech tags are now part of nalaf.

Choice of CRF Library

The modularity of our pipeline allows to easily swap one particular implementation of CRFs for another. However, we had to decide nonetheless which default CRF library we wanted to use. There are a few popular existing libraries: wapiti [12], MALLET [13], CRF++ [14] and CRFSuite [15]. Obviously, a library written in Python would be preferred. Unfortunately, none of these libraries are written in Python so we have to settle for writing wrapper classes to invoke and use them. After careful consideration of the pros and cons of each library we discarded MALLET and CRF++, mostly because the other two libraries perform significantly faster.

One key requirement for us was to be able to input real valued features, or at least

assign real valued weights to the input features. This requirement is due to the word representation features discussed in subsection 3.6.3. Since CRFSuite is the only library that supports such features we picked that implementation as default.

However, one big drawback to the choice of CRFSuite is the inability to utilize multiple cores on a multi-core machine, in comparison to wapiti which can run in multi-threaded mode. The reader should note that if real valued features are not used in a specific task and training time is essential, wapiti might be a better choice.

3.3.3 nala

After the separation, now nala became a specialized tool for mutation mention extraction, using nalaf as library. Modules for generating mutation specific features for example, implement the general interface defined by nalaf and are easily composed together to create a model.

Furthermore, nala includes implementation of a document selection pipeline for the purposes of the bootstrapping process, as well as postprocessing module specific to mutation mentions. Additionally, it includes wrappers around the GNormPlus system [16] for tagging of genes and proteins as well as helper methods for obtaining data from Uniprot.

3.4 Overview of Used Techniques

The following section aims at providing a brief overview of the key techniques used in our method, as well as the motivation behind using them. Most of these approaches are general, however, we view them in the context of named-entity recognition.

3.4.1 Conditional Random Fields

Conditional random fields (CRFs) are an instance of undirected probabilistic graphical model. They are particularly suitable for processing any kind of sequential data [17] such as natural text or images [18] for example. To date, they are probably the most popular method for sequence labeling and parsing tasks (e.g. named entity recognition or part of speech tagging), especially in the biomedical domain [19]. CRFs are able to learn from data by utilizing a set of indicator feature functions that depend on features of a token at position i in the sequence and the labels of the current and previous token l_i and l_{i-1}. Technically, this is a special instance of a CRF model, called a liner-chain CRF, since we are only depending on the current and previous labels. More general CRFs also exist, but inference is often not feasible for such general models. More on the design of typical features can be found in section 3.6.

Most state-of-the art methods for NER used CRFs as their model, including mutation mention methods. With that taken into consideration, along with the ease of training and the fact that the learned model is easily interpretable we also choose CRFs as the basis for our method. The choice behind the particular implementation we used is detailed in subsubsection 3.3.2.

3.4.2 Recurrent Neural Networks

In the last period, neural network (NN) architectures have gained significant attention, mostly in the form of deep and convolution networks and are consistently outperforming traditional machine learning algorithms on variety of tasks. Most state-of-the-art systems for tasks such as image processing or speech recognition for example are based on NN architectures. Similarly, many NLP tasks have seen notable boost in performance using these kinds of models.

One key aspect we have to keep in mind when working with textual data is that we are essentially dealing with sequences (e.g sequence of sentences or words). By default, sequences (especially those of variable length) are not accepted as input for standard feed-forward networks. Typical solution to use sequences in these types of networks is to rely on word embeddings and concatenating or summing over the vectors in the sequence. The main disadvantage of these approaches is that we are disregarding the order of the elements in the sequence, information that is essential to many learning tasks. Convolution approaches also allow to encode sequences of arbitrary length into fixed sized vectors, using various convolutions and pooling strategies [20]. Although they do take into account word order [21], they are limited to modeling only local patterns. Recurrent neural networks (RNNs) [22] overcome that disadvantage by both allowing for sequences of arbitrary length and utilizing the order in the sequence.

To give a simple overview, a simple RNN reads thorough a sequence iteratively, updating its hidden state based on the current element in the sequence and the hidden state from the previous time step. Traditional RNNs however, suffer from the vanishing gradient problem and therefore are difficult to effectively train. Various improvements have been proposed that solve this problem. Additionally, these improvements allow the RNN to adaptively remember of forget information from previous time steps.

One variation is the Long Short Term Memory (LSTM) [23] architecture that uses part of the state representation as memory cells controlled by gating (input, forget and output gates) units. These units essentially decided how much of the content in the memory cell we should remember or forget. Another improvement is the Gate Recurrent Unit (GRU) [24, 25]. In GRU, instead of having the current hidden state depending directly on the previous time step, we used two gates (reset and update gate) that effectively decided how much information from the previous time step is

allowed to go through. Both LSTM and the newly develop GRU use the same principle of gating, however GRU does not have separate memory cells and uses less gates. At the moment most of the state-of-the-art sequence modeling results are based on LSTM models.

3.4.3 Semi-supervised Learning

The performance of NER models, in general, is intrinsically limited by the availability of high-quality-annotated corpora. The construction of such corpora is usually costly and even more so when there is a need for expert annotators. In the biomedical domain, the difficulty of the task is even greater compared to other domains, due do the much higher number of possible named entities, which keeps increasing constantly as new proteins, genes and mutations get discovered. Additionally, when it comes to NL mentions in particular, humans often disagree on what part of the text exactly constitutes the mention. This in turn makes it even more difficult to create corpora with high inter-annotator agreement and slows down the corpus creating process.

One approach to combat the lack of large annotated corpora, is to turn to the exploitation of large volumes of unlabeled text, applying a semi-supervised learning approach. The idea is to use unlabeled data to reduce data sparsity and improve the overall performance of traditional NER models in conjunction with the labeled data. One way to achieve this is with unsupervised feature learning. Some of the assumptions behind such approach is that the data lie on a lower dimensional manifold compared to the input space or that it tends to form clusters. Additionally, we are leveraging the smoothness assumption, or in other words data point close to each other tend to have the same label.

3.4.4 Unsupervised Feature Learning

In general, when we talk about feature learning, we are referring to a set of techniques for transformation of raw data, e.g. pixels in an image or words in a text, into useful representations that can be effectively used for machine learning. This is in contrast to the manual feature engineering, which is often an expensive and time consuming task, where we hand craft features based on domain and expert knowledge.

The key idea behind unsupervised learning is to find a latent representation of the data space. More specifically, when we are working with text data we are trying to learn useful word representation. In this context, word representations are often simply real valued vectors associated with each word in our vocabulary.

There have been a few different approaches to learning word representations. Common early techniques were based on clustering methods such as the popular Brown

clustering algorithm [26] or [27]. The next group of features commonly refereed to as distributional representations include techniques such as latent semantic analysis (LSA) [28], latent Dirichlet allocation (LDA) [29] and random indexing (RI) [30]. And finally we have neural word embedding features, which are based on neural network models.

Word embedding approaches have gained more popularity in the last few years and they have been shown to improve performance in various NLP task. Relatedly, word embeddings improve performance in biomedical NER tasks as shown by [31], [32] and [33]. In terms of the underlying neural network architecture there are two commonly used variations, namely, the CBOW (continuous bag of words) and the skip-gram model [34]. The main difference between these two approaches is how we are creating labels out of the unlabeled data to train the neural network. Briefly, CBOW tries to predict the current input word based on its surrounding context, while skip-gram tries to predict the surrounding words given the current word. Yet another approach is GloVe (Global Vectors for Word Representation) [35], where the training is performed on global word to word co-occurrence statistics.

After we learn word representations, with a certain method, we use them in one of two ways: either as continuous features in the CRF setting described in subsection 3.6.3 or as features in an RNN model described in subsection 3.6.4.

3.4.5 Active Learning

The key idea behind active learning, as a special case of semi-supervised learning, is the ability to interactively query some reliable source of information, often the user, for the true labels of some data points of interest. During the training process of a model, that means that we want to query the labels for data points for which the models is still uncertain about. Effectively, we are lowering the number of examples needed to cover a certain difficult part of the problem space, compared to the traditional supervised learning approach. In other words, we are iteratively guiding the model to quickly learn a certain difficult concept.

Another way to view the active learning approach is: as input we have a model we are trying to learn or improve upon, a small set of labeled data and comparatively large set of unlabeled data, while as output we desire to produce new labeled data by querying an external source.

There are different algorithms for choosing candidate data points to be labeled from the set of all unsupervised points. A general survey can be found in [36]. Some approaches include: sampling points for which the current model has very low confidence, sample points that are expected to reduce generalization error or output variance. Another common approach is query by committee. In this approach, first, different models are trained using different classifiers, then we apply the model on

unlabeled data. Finally, we pick the data points for which the committee disagree most.

Active learning has been applied to many NLP tasks, including NER with great success [37]. In our particular task, the model is usually able to easily capture standard mutation mention but struggles with semi-standard and natural language mention. Therefore, to help the modeling of such mentions we need to do two things: firstly, select documents that are likely to have SS or NL mentions and secondly, manually annotate those documents and add them to the training dataset. In turns out the first task of selecting appropriate documents is not entirely trivial. Our approach is covered in subsection 3.9.1.

3.5 Preprocessing

Preprocessing is an important part of any text mining task. The main goal behind the preprocessing step is to identify meaningful units from a given text that we will process further. Our dataset is a collection of documents, which themselves are a collection of parts. Each part contains portion of the text that needs to be split further down into smaller and more useful units. For the purposes of our task those units are sentences and tokens since we want to extract some useful information, such as features on both sentence and token level. Therefore, the task of splitting a part into sentences and splitting a sentence into tokens are separate modules with their own interface as part of the preprocessing package.

Currently we are using a single implementation of the interface `nalaf.preprocessing.splitters.Splitter`, namely **NLTKSplitter** utilizing the sentence tokenization provided by NLTK [38].

3.5.1 Tokenization

The tokenization is certainly one of the most important parts of the preprocessing step and it can potentially have a big impact on the performance of the final method. Therefore, we explored different tokenizers, all of which implement the interface `nalaf.preprocessing.tokenizers.Tokenizer`

The most basic implementation is the **NLTKTokenizer** which utilizes the word tokenization provided by NLTK [38]. Here simply each word in the sentence becomes a token, with additional steps such as splitting standard contractions or treating most punctuation characters as separate tokens.

The other implementations are variation of the tokenizer used in tmVar [1]. The idea behind the **TmVarTokenizer** is to define tokens on a finer level than words in order to capture individual components that make up a mutation mentions. To achieve that besides splitting words into tokens, special characters such as brackets are their own

tokens. We also split into separate tokens transitions between numbers, uppercase and lowercase letters. For example the mention *'c.(A27G)'* will be split into tokens *['c', '.', 'A', '27', 'G']*.

Building on the original tmVar tokenizer we implemented several variations:

1. Remove splitting into separate tokens transitions from uppercase to lowercase letters (but keep the transition from lowercase to uppercase letters). This change will allows us to keep words starting with a capital letter intact and as a single token, while preserving finer tokenization of the components of mutation mentions. For example the mention *'Novel approach'* would be split into:
   - *['N', 'ovel', 'approach']* before the change.
   - *['Novel', 'approach']* after the change.
2. Split non-ascii characters into separate tokens. For example *'єA411P'* would be split into:
   - *['єA', '411', 'P']* before the change.
   - *['є', 'A', '411', 'P']* after the change.
3. Join repeating non-alphanumerical characters into a single token. For example *'(Gly —> Arg)'* would be split into the tokens:
   - *['(', 'Gly', '-', '-', '-', '>', 'Arg', ')']* before the change.
   - *['(', 'Gly', '—', '>', 'Arg', ')']* after the change.

3.5.2 Labeling

The annotations of entities in most dataset are provided on a mention level, meaning small spans of the text are marked as a certain entity (e.g. mutation mention). That in turn means that one annotation mention might be in fact spanning across several token, depending on which tokenization was used. However, for most learning tasks we need to provide class labels not on a mention level, but rather on a token level. Therefore, we have to come up with a labeling scheme which given a span of text which represents an entity assigns a label to each of its tokens.

The labeling step has its own interface as part of the preprocessing pacakage. We explored and implemented the following labeling schemes:

1. **IOLabeler** uses the IO (inside, outside) format. Creates labels based on the class ID of each entity and the location of the token with regards to the mention span:
   - **I-[class ID]** marks a token inside the mention.

- **O** marks a token outside the mention.

2. **BIOLabeler** uses the BIO (beginning, inside, outside) format. Creates labels based on the class ID of each entity and the location of the token with regards to the mention span:
   - **B-[class ID]** marks the first token of the mention.
   - **I-[class ID]** marks a token inside the mention.
   - **O** marks a token outside the mention.
3. **BIEOLabeler** uses the BIEO (beginning, inside, ending, outside) format. Creates labels based on the class ID of each entity and the location of the token with regards to the mention span:
   - **B-[class ID]** marks the first token of the mention.
   - **I-[class ID]** marks a token inside the mention.
   - **E-[class ID]** marks the last token of the mention.
   - **O** marks a token outside the mention.
4. **TmVarLabeler** is a close approximation of the labeling scheme used by tmVar [1]. Exact replication was not possible because algorithmic details where not provided in their paper, nor was the complementary part of the source code provided. The labels are assigned using a set of regular expression along with simple rules based on the assigned label to the previous token relative to the token being currently labeled. The following 10 labels can be assigned:
   - **A**: Reference sequence.
   - **T**: Mutation type.
   - **F**: Frame shift.
   - **R**: SNP.
   - **M**: Mutant.
   - **W**: Wild type.
   - **S**: Frame shift position.
   - **P**: Mutation position.
   - **I**: Other inside mutation tokens.
   - **O**: Outside.

3.6 Feature Design

The process of feature engineering is a vital part of learning any model. The aim is to create features that can capture well important characteristic useful for the prediction. When it comes to CRFs usually there is a set of features that work well regardless of the problem domain, and an additionally set of features engineered using domain knowledge. We designed and implemented 4 types of different features:

- Standard NER features.
- Mutation specific features.
- Word embedding features.
- Features based on RNNs.

3.6.1 Standard NER Features

These features are the ones typically used in NER task, and perform well regardless of the problem domain. They include:

- `nalaf.features.simple.SimpleFeatureGenerator` generates a feature that is simply text value of the token itself.
- `nalaf.features.simple.SentenceMarkerFeatureGenerator` generates a binary feature that marks the first ('BOS') and the last ('EOS') token in the sentence .
- `nalaf.features.simple.NonAsciiFeatureGenerator` generates a simple binary features indicating whether the token contains non-ascii characters.
- `nalaf.features.parsing.PosTagFeatureGenerator` generates features that mark the part-of-speech tag (POS tag) such as noun, verb, adjective of the token. The idea behind using them is to better capture NL mentions, since most NL mention will follow certain patterns (e.g. a noun followed by a preposition). To generate the POS tags we used the versatile text processing Python library TextBlob [39], which itself in the background uses the fast POS tagger implementation of library pattern [40].
- `nalaf.features.parsing.NounPhraseFeatureGenerator` generates binary features indicating whether the token is a part of a noun phrase and it is implemented using the noun phrase extraction module from TextBlob [39].

- `nalaf.features.conjunction.ConjunctionFeatureGenerator` for a set of already existing features, specified by the user, defines new so called conjunction features. More specifically, the user specifies a list of tuples, where each tuple consists of 2 or more existing features, and this generator simply concatenates the values of the features in the tuple.

- **Orthographic features** such as: word shape features, binary indicators for special characters, number of uppercase, lowercase and numeric characters, 1...5 prefix and suffix characters. Note that currently most of these features are implemented in `nala.features.tmvar.TmVarFeatureGenerator`, although since they are not specific to mutation mentions they will be transfered to nalaf in their own module.

3.6.2 Mutation-specific Features

The next set of features aim to capture information from the domain about the way mutation mentions are typically formed. They exploit the fact that certain keywords will most likely appear such as: mentions of amino acids, words like "substitution", "insertion", "deletion", etc. We implemented there different subtypes of features, two of which are aimed at capturing standard mentions and one aimed at capturing natural language mentions.

- The first set of features generated by `nala.features.tmvar.TmVarFeatureGenerator` are based on the features from the tmVar system [1], and are simple regular expression indicating the presence of different components of mutation mentions such as: DNA or protein symbols, amino acids in full or abbreviated form and keywords indicating the type of mutation, e.g. deletion, duplication, frameshift, etc.

- `nala.features.tmvar.TmVarDictionaryFeatureGenerator` generates the second set of features, also replicated from the tmVar system. The idea behind them is to capture the Human Genome Variation Society HGVS mutation nomenclature [10]. They consist of 11 regular expression 7 of which are used to match genomic mutations and 4 for protein mutations. The regular expressions are executed against each part. Whenever there is a match, the tokens spanning that match are marked with B/I/E (beginning, inside or ending of the match). Tokens not matched by any of the patterns have the value O.

- The third set of features aim at capturing patterns in NL mentions. With `nala.features.regex.RegexNLFeatureGenerator` we try capture mentions expressing deletion mutations with a set of 7 regular expression. Match against any of those

will result in a binary indicator feature. With `nala.features.nl_mutations.SemiStandardFeatureGenerator` we want to capture more general NL mentions. These features are executed on a sentence level, using a combination of regular expression and simple rules. For example if there is match in the sentence for regular expression indicating the position of the mutation and there is a mutation word present (e.g. substituted) in the sentence then we mark the appropriate matched tokens as well as any other mention of an amino acid withing the same sentence.

3.6.3 Word Representation Features

As we already discussed in the section on semi-supervised learning, leveraging large bodies of unlabeled text can potentially improve performance of NER models. Therefore we designed features that use word representations as features for a CRF.

First we trained word embedding using both the CBOW and skip gram model. More specifically we used the word2vec implementation by the gensim [41] Python library. The training was done on the complete MEDLINE/Pubmed database, using the text from abstracts and titles of all the available articles preprocessed with splitting and tokenization. The word embeddings are simply D dimensional real valued vectors. Traditionally CRF model do not support real valued features, to overcome this a CRF feature $emebedding_i$ is formed, where $0 < i < D$, for each dimension, and the numerical value for that dimension is used as a weight for the feature. Additionally, we allow for setting a weight multiplier that will apply to every dimension.

Secondly we implemented features based on clusters induced by the Brown clustering algorithm. Since semantically or syntactically similar words are likely to belong in the same clusters we develop features based on the brown cluster assignment. To induce the clusters from unlabeled data we used Liang's [42] C++ implementation of the algorithm. The Brown clustering algorithm forms hierarchical clusters and induces a binary there. Each word in the corpus is a single leaf, represented as string of 1s and 0s, encoding cluster assignments up to the root of the tree. Then for each word, we generate features representing all sub paths from the root to the word. For example the word 'mutation' is encode as 10111100, and we will generate the features $'1', '10', '101', \ldots, '1011110', '10111100'$ for it.

3.6.4 RNN-based Features

In this section, we described a set of features derived from external models trained on recurrent neural networks (RNNs). For the motivation behind using RNNs for sequence tagging, as well as a brief overview see subsection 3.4.2. Early experiments in training

these RNNs, showed that they are capable of capturing some limited information about mutation mentions. Typically, very large amounts of data are required for achieving good performance in RNN models. Since, we have a rather small sample of labeled, instead of using the RNN models separately, we decided to use their probabilistic output as an input feature in our traditional CRF model.

We constructed several different types of neural network architectures using the keras [43] library built on top of theano[44, 45]. We train our data in the network, setup in layers as following:

1. **Input/Embedding layer** - Here, we explored two variations. Either using the pre-trained word embedding vector representations from all PubMed articles, to transform the textual data directly, or creating an Embedding layer that accepts indices representing words in our vocabulary. In the second approach, we essentially train word representation only from our labeled data, which might make them more suitable to solving the task at hand compared to general word representations.

2. **RNN layer** - Here we explore three different type of RRNs: an LSTM, GRU or a bidirectional RNN with an LSTM as forward pass and a GRU as backwards pass.

3. **Dropout layer** - Optional dropout layer with a probability of 0.5.

4. **TimeDistributedDense layer** - A simple densely connected layer that additionally stretches into the time dimensions (or in other words along the sequence length).

5. **Activation layer** - Using a soft-max activation function.

For the choice of hyper-parameters the following where chosen:

- an embedding layer of 100 dimensions.
- maximum sentence length of 150, longer sequences were truncated, this number was chosen after inspecting the distribution of sequence length for the complete dataset.
- 512 hidden units in both the second and the forth layer.
- batch size of either 32 or 64, note that RNNs are typically more sensitive to the choice of batch size.
- our loss function was categorical cross entropy, fitting using the adam optimizer.
- we used early stopping, on a validation set (20% from the training set) with a patience of 2 epochs.

Finally, the class probability output, for every token, from a certain configuration of a network as described above is used a CRF real valued feature multiplied by a user chosen weight (default 1). Note that there are $2x3x2x1x1 = 12$ different network configurations.

3.7 Postprocessing

Several different methods [1, 46, 47] have show significant increase in performance including some sort of postprocessing module. Usually a postprocessing module refines the predictions obtained from the module. The refinements performed include: boundary extension to correct for partially matched entities, disambiguation between conflicting results if more than one classifier is used, resolution of abbreviations, use of repeating occurrences of the same entity, etc. In our system, postprocessing is defined as a separate module in `nala.learning.postprocessing` enabling easily switchable various implementations. We implemented several different postprocessing strategies and combined them together in one module. They are:

- **Splitting of multiple mentions** - According to our annotation guidelines, if we encounter several mentions close to each other joined by some keyword or character (e.g 'and', 'or', '/'), we annotate them as separate entities. Therefore, we detect the cases where the model predicts one joined mention (e.g. 'c.A27G/c.G33A'), and split them into to separate mentions (e.g.'c.A27G', 'c.G33A'). Since not all mentions that contain the relevant keywords are in fact multiple mentions we perform additional checks on the splits mentions before adding them.

- **Fixing mention boundaries** - Here we aim to correct mentions that are only partially matched due issue with boundaries. We perform several corrections: removing an extraneous parenthesis or adding missing parenthesis at the begin or end of the mention. Additionally, we extend incomplete frame shift mentions at the end. For example the model predicted 'G75fs', but we know that frame shifts typically end in fsX followed by a number. Therefore, we look at the neighboring characters and include them if they fit. In the example we would correct the prediction to 'G75fsX107'. Another boundary correction, tries to detect aa missing '-' or '+' token at the beginning of the mention and includes it if appropriate.

- **Inclusion via existing predictions** In order to catch possible false negative mentions and improve the recall of the model we use the existing prediction and find similar prediction to them. More specifically, for each prediction of the model, we build a regular expression, where each specific number and letter is replaced by a pattern matching any number or letter, while preserving the other special

characters (e.g. +, -, IVS, rs, etc.) and the order. For example if the system predicted '+27C>T', we will form the regular expression '+[0-9][0-9][A-Z]>[A-Z]' and try to find additional matches. An improvement over this strategy was to only consider predictions that have relatively high confidence, and to only consider mentions from the ST subclass.

- **Inclusion via positive patterns**. In another attempt order to catch possible false negative mentions and improve the recall, we defined 18 comprehensive custom regular expression. They are aimed mostly at matching SS and ST mentions and are designed to only match expression for which we are highly certain that they represent a valid mention. If the match does not exist in our set of predictions we simply add it. Otherwise, if we possibly already have an existing overlapping mention and the match has greater length we replace it.

- **Exclusion via negative patterns** Finally, we designed a set of regular expressions aimed at catching false positives in order to improve the precision. For example, often the model would think a mention of a numbered amino acid e.g. 'Val 45' is a mutation, when in reality it is not, and the text was referring to the mention in some other context. Any matches by these so called negative patterns are then removed from the set of predictions.

Experiments with combinations of the above mentioned patterns showed that every one of them increase the overall performance, with the exception of 'Inclusion via existing predictions'. Therefore, our final postprocessing module includes only the other 4 strategies. For details on the performance increase see section 4.6.

3.8 Training Strategies

This section discussed the process of manipulating the training data to overcome shortcomings of the dataset or seed up the development of a model. We named this process training strategies, and explore two different types: strategies based on pruning and strategies based on multiple models.

3.8.1 Pruning

These type of training strategies deal with selecting a subsample of the training data, before sending it to the learning procedure. The necessity of such strategies is two fold. Firstly, during the development of the method we highly valued the ability of quickly trying out new features, pre and post processing steps, or architectures and seeing their effect on the performance. Since training on the complete train set usually

takes significant amount of time, we wanted to train on a smaller subset, there by enabling us to do fast prototyping. The second reason is the data sparsity. In most of our full text document, only a small subset of their parts contain any mentions at all. Furthermore, even for the parts that do contain mentions, they are located in a relatively small number of sentences. Here by parts we mean parts of text from the document, logically belonging together, usually few paragraphs (e.g. the abstract is one part).

For the tmVar corpus only **15.2%** of sentences contain any mention at all. For the IDP4 corpus that number is even smaller at **11.9%**. When we look at token level, things get even worse, only **3.8%** and **3.7%** of tokens are part of a mention (have a label other than 'O') in tmVar and IDP4 respectively. The idea behind the training strategies, in terms of the learning procedure, is to help the model learn better by under-sampling the negative label (label 'O').

To achieve that, we developed two separate training strategies:

- **Simple part pruning** - With this strategy we simply remove the parts that do not contain any mentions from the training data, while keep every other part with at least one mentions.
- **Sentence ratio pruning** - With this strategy we look at individual sentences in each part. All of the sentences that contain at least one mention are kept in the training data. For the rest of the sentences, we randomly sample only P%, where P is hyper-parameter specified by the user. We explored fractions of 0%, 25%, 50%, 75% and 100%.

3.8.2 Multiple Models

As we have already seem, or training dataset distribution is highly skewed towards examples of the negative label. Furthermore if we examine the distribution in term of subclasses for the positive label, we can see that in the IDP4 corpus we have around **91.9%** of standard mentions, **5.9%** semi-standard mentions and **2.1%** natural language mentions. Again, we have highly skewed distribution towards the ST subclass. Additionally, in early experiments during the development of the method our models exhibited fairly good performance on the ST subclass and was struggling with the SS and NL subclasses. Hand designed feature aimed at better capturing those 2 classes had little to no effect since the model was overwhelmed by examples of one class.

To combat these shortcomings we devised a strategy where we train separate models with examples of only a subset of the subclasses. Specifically, first we run the ExclusiveNLDefiner on the training data, then we delete the mentions of a certain subclasses. Note that here we only delete the labels that signify a mention was annotated at a

certain position in the text and not the sentence or the tokens themselves. The subclasses for the separate models where grouped as follows: only ST on one hand, and either only SS and separately only NL as a 3 model variant or a combination SS + NL for 2 model variant. The final predictions are obtained by taking the union from the predictions of the separate models.

3.9 Bootstrapping

In section 2.2 we discussed and concluded that NL mentions are indeed significant. Since they are significant, our goal to create or extend a corpus to include more natural language mentions is justified. This section details how we extend the original IPD4 corpus, named IDP4+, using the principles of active learning explained in subsection 3.4.5. The process, named bootstrapping, was divided in several main subtasks: creation of document selection pipeline, candidate generation, manual annotation and evaluation. The idea is to apply the bootstrapping process in several subsequent iterations, hopefully improving the model after each one.

3.9.1 Document Selection Pipeline

In order to enrich the corpus with more NL mentions first we need to select articles that are likely to have such mentions from the vast body of unlabeled articles. For that purpose we built the document selection pipeline as separate module with its own interface defined in `nala.structures.selection_pipelines.DocumentSelectorPipeline`. Note that although such a pipeline can generally be applied for any NER task, at this point in the development of nalaf and nala it is still tied to task of mutation mentions. Conceptually, a document selector executes a step of filters that act on an initial set of documents to produce a small subset of documents of interest. In our case, documents that are likely contain NL mentions. The execution of the selection pipeline, implemented as chain of Python generators, is as follows:

- **Initialization** - Initializes the pipeline with an initial document selector that selects all possible documents we want to consider. So far, we implemented only one such selector in `nala.bootstrapping.utils.UniprotDocumentSelector`. This selector is a replication of the process used to select documents during the creation of the original IDP4 corpus.

  Briefly, the selector first selects as set of proteins from Uniprot given some query. The default query we used is the following:

  ```
  (annotation:(type:natural_variations) OR annotation:(type:mutagen))
      AND reviewed:yes AND organism:"Homo sapiens (Human) [9606]".
  ```

This translates to selection of all the proteins for human, that are in Swissprot and contain an annotation for either natural variations or mutagenesis. Then, for each of those proteins, we download their xml from Uniprot and look for a so called "evidence" attribute for a sequence variant or mutagenesis site. One type of possible evidence is a PMID for a Pubmed article. Finally, we collect the unique set of PMIDs and yield them as our initial selection.

- **PMID filters** - Next, we apply a series of PMID filters which as input take a stream of PMIDs as yielded by a previous filter or the initial selector and yield a possibly smaller set of PMIDs. A PMID filter implements the interface defined in `nala.bootstrapping.pmid_filters.PMIDFilter`. So far, we have only one implementation that filters our any PMIDs that have been considered in a previous iteration.

- **Download article** - Then, we run a helper class that takes in the stream of PMIDs, downloads the corresponding articles from PubMed and yields a stream od Documents, as defined in our internal data structure. This step, does not do any filtering and only transforms the data.

- **Document filters** - Finally, we apply a series of Document filters which as input take a stream of Documents as yielded and filter them out based on some logic. We have several implementation chained together.

  The **KeywordsDocumentFilter** is a simple filter, that filters our documents that do not contain at least one of a given set of keywords in at least one of their parts. The default keywords we used are 'mutation', 'variation', 'substitution', 'insertion', 'deletion' and 'snp'.

  The **HighRecallRegexDocumentFilter** uses regular expression to first get a list of sentences with possible natural language mentions. Then each possible mention gets compared to tmVar and nala predictions. If there is no overlap, that is a sign that the existing model cannot recognize the mention. That mention will be considered as NL and thus the document will not be filtered out. Ever other document gets filtered out. The regular expression are designed to have high recall regarding NL mentions while not worrying too much about the precision, hence the name.

  The **ManualDocumentFilter** simply queries the user if a document is acceptable or not. The idea is that this filter is used last in the chain, and the user can quickly skim over the contents of the abstract and decide if the document contains and NL mentions. Given that all previous filters did their job properly, the user should not have to go through too many documents to find a set of usable ones.

All of these filter are chained one after another in the order they are described, to form a document selection pipeline. A minor implementation detail, worth noting is that most of the filters used a caching mechanism to speed up the selection process in subsequent iterations.

3.9.2 Candidate Generation

In this phase, we take the documents generated from the document selection pipeline, named **candidates**, and we run the best model we have at that point to obtain predictions. Based on the confidence level for each prediction as reported by the model and user selected threshold we mark them as either 'selected' if the confidence is above the threshold or 'pre-selected' otherwise. The idea behind this distinction, is that the user should not pay much attention to mentions marked as 'selected', since the model is already confident about them, but should definitely review those marked as 'pre-selected'. The initial threshold is chosen as 1, effectively marking all mentions as 'pre-selected', with the idea of lowering the threshold in later iterations as the model becomes better and better at predicting NL mentions.

From our earlier experience in annotating mutation mentions during the creation of the IDP4 corpus, we know that it is easier to annotate mutations if you already know the proteins they are associated with. But the manual annotation of then proteins themselves is even more time consuming, especially if the annotator does not have a biomedical background. Therefore, we also run the GNormPlus tagger [16] on the candidates, to obtain predictions for genes and proteins.

3.9.3 Manual Annotation and Evaluation

The final step, after we have the candidates along with the predictions is to manually review them, correcting any false positive mentions or add missing false negative mentions. This is done using tagtog [48], the same platform used to create the original IDP4 corpus. After we have done the manual annotation, we import the reviwed document back into nala and evaluate that particular iteration. We perform two evaluations. First, we evaluate the how good is the method only on the document from that particular iteration. Second, we aggregate all the documents from all previous iterations along with the current one, including the IDP4 corpus (dubbed iteration O), and perform cross validation.

4 Results and Discussion

This chapter details the results of our experiments and outlines the key insights we have gained from them. First, we briefly describe our experimental setup. Then, we show detailed evaluation results for different components of our model including: the tokenizer, labeler and postprocessing module. We also evaluate different training and merging strategies. Finally, we evaluate our performance for each iteration of the bootstrapping process and give an estimation of the standard error of our best model.

In order to accurately calculate the performance of our method we have to establish a sound strategy for dividing corpora into separate sets for training and testing.

The tmVar corpus already comes split into a train and a test set. All of our models were trained only on the tmVar train set and we report performance only on their test set. This, ensures the comparability of our method to the tmVar system.

For the IDP4/IDP4+ corpus we used few different splitting strategies. Early on, during the development of the method we used a simple randomized split into 2/3 of the documents for training and 1/3 for testing. However, due to the highly skewed distribution of subclasses, as well as the uneven distribution of mentions per document, we quickly realized this is not entirely reliable. Therefore, we implemented a stratified split strategy, where we enforce the additional constraint on both the train and test set to have roughly the same distribution of subclasses. The second strategy used was the traditional n-fold cross validation, where we used in most of our experiments values for n of 5 or 10.

Furthermore, during all of our experiments in training the RNN architectures we used a validation set obtained by randomly sampling 20% of the training set. This validation set was used for early stopping of the training of a neural network with a patience parameter of 2. This means that after each epoch we monitor the loss on the validation set as compared to the loss of the previous epochs. If the loss does not improve after N epochs, where N is the patience parameter we stop the training.

An implementation detail worth nothing is that we make sure to always have the same random seed during the randomization part of the splitting to ensure that our experiments are repeatable and the same configuration of hyper parameters will lead to the same results.

Most of the performance measures reported on the IPD4/IDP4+ corpus in the rest of this chapter are calculated using the stratified split strategy, unless noted otherwise.

4.1 Tokenizers

Here we show the performance of different tokenizers as described in subsection 3.5.1. Although we explored the effect of the NLTKTokenizer, the original tmVarTokenizer, as well as different combinations of our improvements over the tmVarTokenizer, there was no significant difference between most of them. In Table 4.1 we show the detailed performance for 3 separate implementations.

		tmVar + non capital + non-ascii			NLTK			tmVar		
class	e/o	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score
ST	e	0.8738	0.8632	0.8685	0.8838	0.8640	0.8738	0.8671	0.8629	0.8650
NL	e	0.2791	0.1277	0.1752	0.3824	0.1368	0.2016	0.3404	0.1684	0.2254
SS	e	0.2308	0.1154	0.1538	0.2000	0.0455	0.0741	0.5000	0.2414	0.3256
all	e	0.8537	0.8132	0.8329	0.8715	0.8149	0.8422	0.8495	0.8155	0.8321
ST	o	0.9563	0.9327	0.9443	0.9662	0.9294	0.9475	0.9478	0.9307	0.9392
NL	o	0.8095	0.4513	0.5795	0.8571	0.3853	0.5316	0.7460	0.4273	0.5434
SS	o	0.4667	0.2500	0.3256	0.2000	0.0455	0.0741	0.6000	0.3000	0.4000
all	o	0.9632	0.9124	0.9371	0.9738	0.9020	0.9365	0.9580	0.9128	0.9349

Table 4.1: Comparison of the performance of tokenizers on the IDP4+ corpus. Three implementation compared. First, original tmVar implementation including the improvement of not splitting by capital letters and separating non-ascii characters into separate tokens. Second, the simple tokenization provided by NLTK. Third, original tmVar implementation.
Postprocessing was included. Performance in terms of the different subclasses shown by the first column. Both exact (e) and overlapping (o) mention level scores shown.

As we can see from Table 4.1 the choice of a tokenizer does make some difference. For example, models using the NLTK tokenizer have significantly more trouble in handling SS mentions. The ST class on the other hand is handled equally well by models using any of the tokenizer. After evaluating all of the tokenizers we chose our default tokenizer to be the original tmVar tokenizer with the two additional improvements. The choice of not splitting by capital letter, makes sense since that enables us to keep words intact, needed to properly employ the word embedding features later on. On the other hand, the splitting of non-ascii characters into separate tokens is kept just for consistency with the idea of having a fine grained tokenization.

4.2 Labelers

The tmVar method reported significant difference in using different labels with F_1 score of 83.24%, 83.82% and 87.67% for BIO, BIEO and tmVar labeling schemes respectively, gaining a noticeable boost in performance from their fine grained labels [1]. Similarly, we expected same kind of boost in performance. However, as you can see from Table 4.2 our scores are much closer to each other across different labels, with the exception of the simple IO labeling scheme which was not used by the tmVar system at all.

	BIEO			BIO			IO			tmVar		
e/o	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score
e	0.9463	0.9116	0.9286	0.9244	0.8966	0.9103	0.9095	0.8645	0.8864	0.9353	0.9030	0.9189
o	0.9871	0.9523	0.9694	0.9791	0.9611	0.9700	0.9813	0.9516	0.9662	0.9894	0.9570	0.9729

Table 4.2: Comparison of the performance of labelers on the tmVar corpus. Four implementations compared. First, labeling using the BIEO (begin, inside, end, outside) format. Second, labeling using the BIO (begin, inside, outside) format. Third, labeling using the IO (inside, outside) format. Forth, labeling with the replication of tmVar's fine grained labels.
Postprocessing was included. Both exact (e) and overlapping (o) mention level scores shown.

We observe similar behavior with the IDP4+ corpus as seen in Table 4.3. The only difference here is that the simpler IO labeling seem to work as well as the others in general and slightly better for the NL and SS classes. This might be due to the higher variability of unique tokens as part of NL and SS mentions compared to the variability of tokens in ST mentions.

4.3 Merging Strategies

In subsection 3.1.2 we described a general merging scheme for merging the annotations from multiple annotators in the IDP4 corpus. In step 1 of the scheme, we can choose between 2 general merging strategies, and in step 2 of the scheme we can choose between between 3 partially overlapping entity merging strategies, amounting to total of 6 combinations. We explored the effect on performance of all 6 combinations. Additionally, we explored only using a subset of annotators, instead of all 4. However those comparisons showed no significant difference and are not shown here.

In Table 4.4 we show the performance of the intersection merging strategy in combination with all 3 entity merging strategies. While in Table 4.5 We show the performance of the union merging strategy in combination with all 3 entity merging strategies.

		BIEO			BIO			IO			tmVar		
class	e/o	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score
ST	e	0.8738	0.8632	0.8685	0.8634	0.8655	0.8645	0.8493	0.8375	0.8434	0.8785	0.8583	0.8683
NL	e	0.2791	0.1277	0.1752	0.2708	0.1383	0.1831	0.2034	0.1263	0.1558	0.2368	0.0947	0.1353
SS	e	0.2308	0.1154	0.1538	0.4286	0.2222	0.2927	0.2857	0.2143	0.2449	0.2727	0.2308	0.2500
all	e	0.8537	0.8132	0.8329	0.8433	0.8172	0.8300	0.8200	0.7895	0.8045	0.8559	0.8081	0.8313
ST	o	0.9563	0.9327	0.9443	0.9449	0.9348	0.9398	0.9577	0.9340	0.9457	0.9641	0.9328	0.9482
NL	o	0.8095	0.4513	0.5795	0.7612	0.4513	0.5667	0.7229	0.5042	0.5941	0.8214	0.4035	0.5412
SS	o	0.4667	0.2500	0.3256	0.4286	0.2222	0.2927	0.3636	0.2759	0.3137	0.2727	0.2308	0.2500
all	o	0.9632	0.9124	0.9371	0.9514	0.9144	0.9325	0.9615	0.9212	0.9409	0.9678	0.9089	0.9375

Table 4.3: Comparison of the performance of labelers on the IDP4+ corpus. Four implementations compared. First, labeling using the BIEO (begin, inside, end, outside) format. Second, labeling using the BIO (begin, inside, outside) format. Third, labeling using the IO (inside, outside) format. Forth, labeling with the replication of tmVar's fine grained labels.
Postprocessing was included. Performance in terms of the different subclasses shown by the first column. Both exact (e) and overlapping (o) mention level scores shown.

The general conclusions that can be drawn by examining Table 4.4 and Table 4.5 are the following:

- Intersection is consistently outperforming union with an average difference of 2%-5%, depending on the entity merging strategy. This makes a lot of sense, since we are effectively only considering mentions for which all of the annotators agree

		shortest entity			longest entity			priority		
class	e/o	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score
ST	e	0.8957	0.8875	0.8916	0.8502	0.7790	0.8130	0.8738	0.8632	0.8685
NL	e	0.2927	0.1379	0.1875	0.1746	0.1100	0.1350	0.2791	0.1277	0.1752
SS	e	0.3125	0.1562	0.2083	0.0385	0.0370	0.0377	0.2308	0.1154	0.1538
all	e	0.8756	0.8376	0.8562	0.8081	0.7272	0.7655	0.8537	0.8132	0.8329
ST	o	0.9420	0.9322	0.9371	0.9465	0.8722	0.9078	0.9563	0.9327	0.9443
NL	o	0.7966	0.4519	0.5767	0.6667	0.4715	0.5524	0.8095	0.4513	0.5795
SS	o	0.4118	0.2121	0.2800	0.2414	0.2333	0.2373	0.4667	0.2500	0.3256
all	o	0.9442	0.9071	0.9252	0.9445	0.8623	0.9015	0.9632	0.9124	0.9371

Table 4.4: Comparison of the performance of intersection merging strategy on the IDP4+ corpus. Three entity merging strategies compared, show from left to right, shortest entity, longest entity and priority.
Postprocessing was included. Performance in terms of the different subclasses shown by the first column. Both exact (e) and overlapping (o) mention level scores shown.

to some degree. Edge cases, for which some annotators would agree that they constitute a mention while others would not, are discarded.

- The priority merging strategy is also consistently better than shortest and longest entity for both intersection and union. Again, this makes sense since the priority was assigned according to the highest average inter-annotator agreement as discussed in subsection 3.1.2.

Therefore, we choose the intersection + priority combination for our default merging strategy which is the best performing combination out of all 6.

4.4 Training Strategies

In section 3.8 we described different training strategies and the motivation behind them. In this section we explore the effect they have on performance.

4.4.1 Pruning

First, we perform a comparison of pruning strategies as described in subsection 3.8.1. We explored 7 different pruning strategies: no pruning at all, simple part pruning, and 5 different sentence ratio pruning strategies with the fractions of sentences that do not contain mentions set at 0%, 25%, 50%, 75% and 100%. The training strategies

		shortest entity			longest entity			priority		
class	e/o	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score
ST	e	0.8463	0.7957	0.8202	0.8763	0.7901	0.8310	0.8405	0.8118	0.8259
NL	e	0.3667	0.2136	0.2699	0.2769	0.1525	0.1967	0.2879	0.1583	0.2043
SS	e	0.2759	0.1311	0.1778	0.1860	0.1569	0.1702	0.4000	0.1754	0.2439
all	e	0.8195	0.7418	0.7787	0.8334	0.7313	0.7790	0.8144	0.7536	0.7828
ST	o	0.9138	0.8646	0.8885	0.9342	0.8493	0.8898	0.9269	0.8861	0.9061
NL	o	0.6933	0.4407	0.5389	0.7273	0.4539	0.5590	0.6628	0.4130	0.5089
SS	o	0.4375	0.2188	0.2917	0.5000	0.4333	0.4643	0.5185	0.2373	0.3256
all	o	0.9156	0.8393	0.8758	0.9332	0.8345	0.8811	0.9272	0.8547	0.8895

Table 4.5: Comparison of the performance of union merging strategy on the IDP4 corpus. Three entity merging strategies compared, show from left to right, shortest entity, longest entity and priority.
Postprocessing was included. Performance in terms of the different subclasses shown by the first column. Both exact (e) and overlapping (o) mention level scores shown.

involving pruning showed no significant difference as can been seen in Table 4.6. The only advantage of using pruning is to reduce the training time.

		25% sentence pruning			75% sentence pruning			simple part pruning			no pruning		
class	e/o	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score
ST	e	0.8411	0.8686	0.8546	0.8677	0.8661	0.8669	0.8738	0.8632	0.8685	0.8960	0.8603	0.8778
NL	e	0.3125	0.1613	0.2128	0.2609	0.1277	0.1714	0.2791	0.1277	0.1752	0.2821	0.1170	0.1654
SS	e	0.1905	0.1429	0.1633	0.2500	0.1481	0.1860	0.2308	0.1154	0.1538	0.2727	0.1200	0.1667
all	e	0.8191	0.8200	0.8195	0.8456	0.8160	0.8306	0.8537	0.8132	0.8329	0.8772	0.8104	0.8425
ST	o	0.9256	0.9401	0.9328	0.9516	0.9361	0.9438	0.9563	0.9327	0.9443	0.9733	0.9250	0.9485
NL	o	0.7761	0.4685	0.5843	0.7910	0.4649	0.5856	0.8095	0.4513	0.5795	0.8070	0.4144	0.5476
SS	o	0.4167	0.3226	0.3636	0.4444	0.2759	0.3404	0.4667	0.2500	0.3256	0.5385	0.2593	0.3500
all	o	0.9346	0.9232	0.9289	0.9589	0.9178	0.9379	0.9632	0.9124	0.9371	0.9793	0.9029	0.9395

Table 4.6: Comparison of the performance of pruning merging strategy on the IDP4+ corpus. Four pruning strategies compared. First, 25% of sentences that do not contain any mention kept. Second, 75% of sentences that do not contain any mention kept. Third, simple part pruning. Forth, no pruning at all. Postprocessing was included. Performance in terms of the different subclasses shown by the first column. Both exact (e) and overlapping (o) mention level scores shown.

4.4.2 Multiple models

Next, we perform a comparison of training strategies in terms of multiple models as described in subsection 3.8.2. We explored the performance of 4 different models: either training on subclasses ST, SS and NL separately, or training with SS+NL joined. The performance of the separate models is shown in Table 4.7.

Merging the predictions of the separate models together, to produce final predictions showed no significant difference compared to the same model and training with all subclasses included (comparison not shown here).

As we can see from Table 4.7 the model trained only on ST instances showed slightly worse performance compared to a general model, but still was fairly close. Interestingly enough we have a few false positive predictions of subclass NL and none of subclass SS.

The model trained only on NL mentions had no improvements compared to the general model and only gives predictions within its own subclass without false positives for the other subclasses. The model trained only on SS instances performed the worst and in the exact matching scenario the difference is not noticeable with the general method. This model also predicts false positives from the NL subclass.

Finally, the model trained on both SS+NL mentions together outperforms the standalone models. Although it is slightly better for those subclasses compared to the

		trained on ST			trained on NL			trained on SS			trained on SS+NL		
class	e/o	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score	Precision	Recall	F_1 score
ST	e	0.8414	0.8078	0.8243	0.0000	nan	nan	0.0000	nan	nan	/	/	/
NL	e	0.0000	nan	nan	0.2292	0.1447	0.1774	/	/	/	0.2667	0.1500	0.1920
SS	e	/	/	/	0.0000	nan	nan	0.4000	0.0625	0.1081	0.2500	0.1212	0.1633
all	e	0.8402	0.8078	0.8237	0.1897	0.1447	0.1642	0.3333	0.0625	0.1053	0.2623	0.1416	0.1839
ST	o	0.9388	0.9067	0.9225	0.0000	nan	nan	0.0000	nan	nan	/	/	/
NL	o	0.0000	nan	nan	0.7121	0.5000	0.5875	/	/	/	0.7812	0.5051	0.6135
SS	o	/	/	/	0.0000	nan	nan	0.4000	0.0625	0.1081	0.5263	0.2778	0.3636
all	o	0.9377	0.9067	0.9219	0.6184	0.5000	0.5529	0.3333	0.0625	0.1053	0.8111	0.5177	0.6320

Table 4.7: Comparison of training on separate models on the IDP4+ corpus. Four different models where trained show from left to right training: just on subclasses ST, just on subclasses SS, just on subclasses NL and finally training on subclasses SS+NL joined together. A value of '/' indicates that no predictions for that subclass where made.
Postprocessing was not included, since the postprocessing module works on all subclasses at once. Performance in terms of the different subclasses shown by the first column. Both exact (e) and overlapping (o) mention level scores shown.

general model shows no significant improvement.

4.5 Features

During the early development stages of nala, we determined the baseline feature set that reliably predicts mutation mentions. This baseline feature set, also referred to as the default features consists of the following generators:

- `SimpleFeatureGenerator`, `PorterStemFeatureGenerator` and `SentenceMarkerFeatureGenerator` as standard NER features.
- `TmVarFeatureGenerator` and `TmVarDictionaryFeatureGenerator` as mutation-specific features.
- `WindowFeatureGenerator` with a window of 6 tokens, 3 from left and 3 from right of the current token and the following subset of features included in the window: the word, the stem of the word, and all of the features generated by `TmVarDictionaryFeatureGenerator`.

After we established our baseline feature set, all newly developed features were compared against it. The rest of this section details the insights and results we gained.

4.5.1 Standard Features

The `NonAsciiFeatureGenerator` made absolutely no difference in the performance. Originally, we noticed that the model had difficulties predicting some mentions that contained some non-ascii characters. The idea was to help the model learn such mentions more easily with this feature. Ultimately the prevalence of these mentions is low, and this feature was not relevant.

`PosTagFeatureGenerator` and `NounPhraseFeatureGenerator` which are typically included features in many NER models, did not make a significant difference in performance. We believe this is mostly due to our fine grained tokenization. Most of the tokens that are part of ST mentions would not be categorized as any specific part-of-speech tag or noun phrase, so the result for ST mentions is not surprising. However, our expectations that these features would help in the case of SS and NL mention were not met. This might be partly due to the insufficient examples of such mentions, so they have to be reevaluated after more examples are gathered.

As input to the `ConjunctionFeatureGenerator` we experimented with different sets of tuples. Specifically we created three candidate lists, and then for each list we generated all combinations of pairs and all combinations of triples. The first list included: the token itself in a window of 2 and the part-of-speech tag also in a window of 2. So for example one possible conjunction can be: `'word[-1] | tag[2] | word [-2]'= 'of | NN | mutation'`. The second list contained the features generated by `SemiStandardFeatureGenerator` in a window of 2. The third list contained the features generated by `RegexNLFeatureGenerator` again in a windows of 2. Unfortunately, none of these feature helped in increasing the performance of our model. On the contrary, the conjunction features based on features aimed at capturing SS and NL mentions made the performance worse. Examination of the learned weights led us to believe that by including these features our model was prone to over-fitting.

Furthermore, using the above mentioned NL and SS mutation-specific feature generators as either standalone features or with different window selections, did not improve the performance, but it also did not make the performance worse as was the case when they were used as conjunctions.

4.5.2 Word Representation Features

We evaluated two types of word representation features: based on Brown clustering and based on neural word embeddings.

The representations induced by Brown clustering showed no significant difference for different settings of the number of clusters (we tried: 100, 500 and 1000 clusters) and different settings of the weight parameter. However, close examination of the

learned weights by the model when using these features showed that a few of them had really high weights, meaning that they are quite descriptive. With that in mind, the reason as to why they did not improve the performance is probably because the same information is already being captured by the hand-crafted features.

When it comes to the word representation features based on neural word embeddings we evaluated both the CBOW (continuous bag of words) and the skip-gram architecture, with context windows of either 5 or 10, trained on the entire set of MEDLINE/PubMed abstracts. Additionally, we experimented with different settings of the weight parameter as described in subsection 3.6.3. Most of the experiment yielded no significant increase in performance with the exception of one configuration. Using the CBOW architecture, with a context window of 10 and the weight parameter at 2 we observed a slight increase in performance.

As we can in Table 4.8, we have an increase of the F_1 score of around 4% for NL mentions and 2% for SS mention in the exact matching scenario. When we allow for overlapping matches there is only an increase of the F_1 score for NL mentions of around 4%. ST mentions do not see any significant change in performance. This is to be expected since the word embedding features are aimed mostly at capturing NL mentions.

Since these features showed improvement we include them along with our baseline feature set as part of our best performing model. Estimation about the standard error for the best performing model, as well as performance evaluation on both tmVar and the IDP4+ corpus can be seen in section 4.8.

A separate experiment in which we included only word embedding and none of the hand-crafted features yielded performance almost as good as the baseline features. This experiment, tells us that the word embeddings definitely captured some vital information to the learning task.

Finally, all of the RNN-based features showed no increase in performance. However, due to time limitations we only performed a small number of experiments and we cannot count these features out as not relevant until more detailed study is performed.

4.6 Postprocessing

In this section we showed the effect of postprocessing on performance as described in section 3.7. The tmVar system also used a form of postprocessing and for them the increase in performance was significant from 87.67% to 91.39%. As we can see from Table 4.9, we have a similar increase in performance, which is two fold, both in precision and recall, and therefore a general increase in F_1 score as well. More specifically, from 88.00% to 92.86% in the exact matching scenario and from 94.67% to

		with out word embeddings			with word embeddings		
subclass	e/o	Precision	Recall	F_1 score	Precision	Recall	F_1 score
ST	e	0.8738	0.8632	0.8685	0.8730	0.8656	0.8693
NL	e	0.2791	0.1277	0.1752	0.3125	0.1613	0.2128
SS	e	0.2308	0.1154	0.1538	0.2222	0.1481	0.1778
all	e	0.8537	0.8132	0.8329	0.8504	0.8177	0.8337
ST	o	0.9563	0.9327	0.9443	0.9552	0.9338	0.9444
NL	o	0.8095	0.4513	**0.5795**	0.8116	0.4956	**0.6154**
SS	o	0.4667	0.2500	0.3256	0.4000	0.2759	0.3265
all	o	0.9632	0.9124	0.9371	0.9625	0.9182	0.9399

Table 4.8: Performance effect of using word emeddings on the IDP4+ corpus. The first column show the performance without word embeddings. The second column show the performance with word embeddings. The CBOW (continuous bag of words) architecture, with a context window of 10 words and a weight of 2 was used to generate the word embeddings.
Default tokenizer and BIEO labeler are used. Both exact (e) and overlapping (o) mention level scores are shown.

96.94% in the overlapping matching scenario. The increase in precision, especially in the exact matching scenario is mostly due to the part of the postprocessing dedicated to fixing boundary issues. When boundaries are fixed entities previously considered a partial match are now matched exactly. The negative patterns, aimed at detecting false positives also help with precision. The increase in recall however, is mostly due to the positive patterns that help catch false negatives.

	without postprocessing			with postprocessing		
e/o	Precision	Recall	F_1 score	Precision	Recall	F_1 score
e	0.9083	0.8534	0.8800	0.9463	0.9116	0.9286
o	0.9742	0.9207	0.9467	0.9871	0.9523	0.9694

Table 4.9: Effect of postprocessing on performance on the tmVar corpus. First, on the left shown the performance without postprocessing, next on the right, the performance with postprocessing.
Default feature set and tokenizer used with BIEO labeler. Both exact (e) and overlapping (o) mention level scores shown.

Similarly to the tmVar corpus our postprocessing module shows increased in performance for the ID4P+ corpus as well, as we can see from Table 4.10. The increase in

performance here is mostly for the ST subclass, which is especially evident in the exact matching scenario. However, since the the ST subclass is the biggest one by far that increase is naturally reflected in the overall increase in performance.

		without postprocessing			with postprocessing		
class	e/o	Precision	Recall	F_1 score	Precision	Recall	F_1 score
ST	e	0.8463	0.7763	0.8098	0.8738	0.8632	0.8685
NL	e	0.2453	0.1354	0.1745	0.2791	0.1277	0.1752
SS	e	0.1333	0.0645	0.0870	0.2308	0.1154	0.1538
all	e	0.8193	0.7291	0.7716	0.8537	0.8132	0.8329
ST	o	0.9620	0.8804	0.9194	0.9563	0.9327	0.9443
NL	o	0.7333	0.4741	0.5759	0.8095	0.4513	0.5795
SS	o	0.4444	0.2353	0.3077	0.4667	0.2500	0.3256
all	o	0.9613	0.8627	0.9093	0.9632	0.9124	0.9371

Table 4.10: Effect of postprocessing on performance on the IDP4+ corpus. First, on the left shown the performance without postprocessing, next on the right, the performance with postprocessing.
Default feature set and tokenizer used with BIEO labeler. Performance in terms of the different subclasses shown by the first column. Both exact (e) and overlapping (o) mention level scores shown.

4.7 Bootstrapping

In order to evaluate if the bootstrapping iterations, as described in section 3.9, are helping with the performance we performed 10-fold cross validation at each iteration i, including the data from all previous iterations $0, 1, \ldots, i_{-1}, i$. We calculate precision, recall and F_1 score for each subclass separately and for every mention in total.

In Figure 4.1 we show the evolution of precision, recall and F_1 score for each iteration, for all mentions together. We can see that both the precision and recall are slowly rising, leading to an increase in F_1 score as well.

More importantly we want to see how the performance changes in terms of each subclass, which is shown in Figure 4.2. The performance of the standard mentions stays relatively unchanged throughout iterations as expected, both since we are not adding many examples of such mentions and because we are already fairly good at predicting them. We have most likely reached the limit for the ST subclass. The

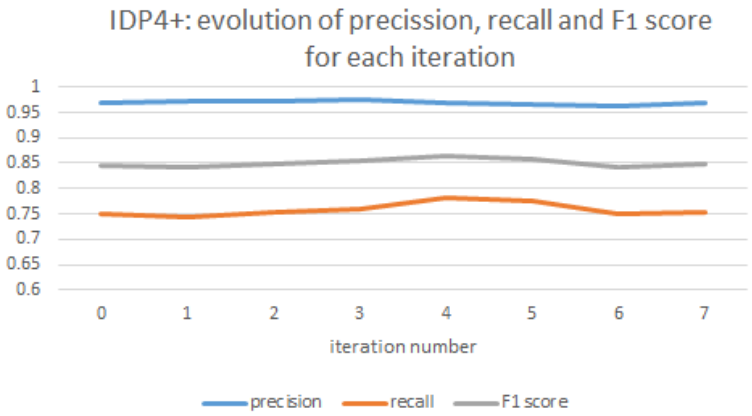


Figure 4.1: Evolution of precision, recall and F_1 score for each iteration for all mentions together. IDP4+ corpus, 10-fold cross validation.

semi-standard subclass shows some improvement during the iterations, but it is still our worse performing subclass. This is naturally to be expected with the smallest number of examples available at around 2%. Finally, the natural language mentions subclass show a slower but steadier increase in performance. This we believe is a direct reflection of the fact that we selected on purpose documents containing NL mentions for every iteration.

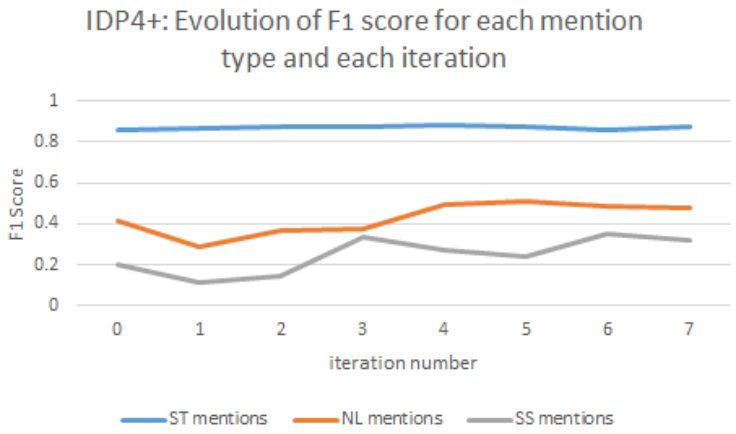


Figure 4.2: Evolution of F_1 score for each mention type and each iteration. IDP4+ corpus, 10-fold cross validation.

4.8 Standard Error Estimation

To truly evaluate the performance of our best model we perform an estimation of the standard error with the following procedure. We randomly select 15% of documents without replacement from the dataset over 1000 iterations. For each iteration we calculate and store the performance measures of interest, specifically: precision, recall and F_1 score. Finally, we estimate the standard error from the 1000 measurements using the standard deviation from the overall performance.

Here, by our best performing model we refer to the following configuration of hyper parameters: baseline feature set + word embedding features, default tokenizer, BIEO labels with simple part pruning and postprocessing included. The results for the IDP4+ corpus are shown in Table 4.11, and the results for the tmVar corpus are shown in Table 4.12.

subclass	e/o	Precision	Recall	F_1 score
ST	e	0.8730 ±0.0027	0.8656 ±0.0025	0.8693 ±0.0025
NL	e	0.3125 ±0.0078	0.1613 ±0.0039	0.2128 ±0.0047
SS	e	0.2222 ±0.0089	0.1481 ±0.0086	0.1778 ±0.0112
all	e	0.8504 ±0.0027	0.8177 ±0.0026	0.8337 ±0.0025
ST	o	0.9552 ±0.0008	0.9338 ±0.0018	0.9444 ±0.0011
NL	o	0.8116 ±0.0061	0.4956 ±0.0059	0.6154 ±0.0053
SS	o	0.4000 ±0.0108	0.2759 ±0.0092	0.3265 ±0.0084
all	o	0.9625 ±0.0007	0.9182 ±0.0018	0.9399 ±0.0011

Table 4.11: Standard error estimation for the IDP4+ corpus for the best performing model with the following configuration: baseline feature set + word embedding features, default tokenizer, BIEO labels with simple part pruning and postprocessing included.

e/o	Precision	Recall	F_1 score
e	0.9487 ± 0.0011	0.9159 ± 0.0017	0.9320 ± 0.0013
o	0.9871 ± 0.0005	0.9543 ± 0.0012	0.9704 ± 0.0007

Table 4.12: Standard error estimation for the tmVar corpus for the best performing model with the following configuration: baseline feature set + word embedding features, default tokenizer, BIEO labels with simple part pruning and postprocessing included.

5 Conclusion and Future Work

By the end of this thesis we can say the we have accomplished all of the goals we set for ourselves in the beginning. We accomplished our first goal of studying the significance of natural language (NL) mentions and showed that they are indeed significant. Leveraging active learning approaches we also accomplished our second goal.

Our method was able to satisfy the requirements of being comparable or better than tmVar. The performance on the tmVar corpus, using our best model is surpassing that of the original tmVar system [1]. We achieve precision of $94.87 \pm 0.11\%$, recall of $91.59 \pm 0.15\%$ and F_1 score of $93.20 \pm 0.12\%$, compared to a precision of 91.38%, recall of 91.40% and F_1 score of 91.39% for the tmVar system [1]. Finally, we can also confidently state that we achieved our goal of creating a tool that will be useful for the community.

All in all, the first notable contribution of this thesis is the study of the significance on natural language (NL) mentions. We show that a significant amount of documents (between 32% and 45% depending on the corpus) contain potentially novel NL mutation mentions not expressed as standard mentions within the same text. Additionally, between 5% and 12% of total mentions are potentially novel NL mentions.

The second notable contribution is the development of nalaf, a well-documented and extensible Python framework for general-purpose text mining tasks. Currently nalaf supports named-entity recognition and relationship extraction, including all pre- and post-processing steps as well as a rich set of feature generators.

The most important contribution is the development of the nala machine learning method for mutation mention extraction. Nala is also available as an open source Python library, is built on top of nalaf and follows the same design principles.

The nala method outperforms current state-of-the-art methods for mutation mention extraction for mentions expressed in standard language. Additionally, nala is able to recognize mentions expressed in natural language, previously unconsidered by any other method with precision of 0.8116 ± 0.0061, recall of 0.4956 ± 0.0059 and F_1 score of 0.6154 ± 0.0053. On the task of recognition of NL mentions there is still room for improvement. The lack of training data for those mentions is clearly reflected in the performance. Our studies suggest that the unsupervised pre-training of word representation features does improve performance on NL mentions. These initial promising results call for further research in this area.

Finally, a by-product of the nala method is the extended IDP4+ corpus. Using an active learning approach, in a series of 7 iterations, we enriched the original IDP4 corpus with new documents containing a significant amount of NL mentions. Our corpus joins other large gold-standard and high-quality corpora for the benefit of the biomedical natural language processing community.

Future work includes additional studies on the neural word embeddings and the RNN-based features as well as improvement of the document selection pipeline. Advanced techniques for comprehensive feature selection have to also be developed. Beyond the scope of this thesis we continue to extend the IDP4+ corpus with more bootstrapping iterations.

List of Figures

List of Tables

Bibliography

[1] C.-H. Wei, B. R. Harris, H.-Y. Kao, and Z. Lu. "tmVar: a text mining approach for extracting sequence variants in biomedical literature." In: *Bioinformatics* (2013), btt156.

[2] J. M. Cejuela, A. Bojchevski, R. Bekmukhametov, S. Karn, and S. Mahmuti. *IDP4 Corpus*. 2015.

[3] L. Hunter and K. B. Cohen. "Biomedical language processing: what's beyond PubMed?" In: *Molecular cell* 21.5 (2006), pp. 589–594.

[4] A. M. Cohen and W. R. Hersh. "A survey of current work in biomedical text mining." In: *Briefings in bioinformatics* 6.1 (2005), pp. 57–71.

[5] W. Hersh. "Evaluation of biomedical text-mining systems: lessons learned from information retrieval." In: *Briefings in bioinformatics* 6.4 (2005), pp. 344–356.

[6] P. Zweigenbaum, D. Demner-Fushman, H. Yu, and K. B. Cohen. "Frontiers of biomedical text mining: current progress." In: *Briefings in bioinformatics* 8.5 (2007), pp. 358–375.

[7] J. G. Caporaso, W. A. Baumgartner, D. A. Randolph, K. B. Cohen, and L. Hunter. "MutationFinder: a high-performance system for extracting point mutation mentions from text." In: *Bioinformatics* 23.14 (2007), pp. 1862–1865.

[8] L. I. Furlong, H. Dach, M. Hofmann-Apitius, and F. Sanz. "OSIRISv1. 2: a named entity recognition system for sequence variants of genes in biomedical literature." In: *BMC bioinformatics* 9.1 (2008), p. 84.

[9] P. Thomas, T. Rocktäschel, Y. Mayer, and U. Leser. *SETH: SNP Extraction Tool for Human Variations*. `http://rockt.github.io/SETH/`. 2014.

[10] P. E. Taschner and J. T. Den Dunnen. "Describing structural changes by extending HGVS sequence variation nomenclature." In: *Human mutation* 32.5 (2011), pp. 507–511.

[11] K. Verspoor, K. B. Cohen, A. Lanfranchi, C. Warner, H. L. Johnson, C. Roeder, J. D. Choi, C. Funk, Y. Malenkiy, M. Eckert, et al. "A corpus of full-text journal articles is a robust evaluation tool for revealing differences in performance of biomedical natural language processing tools." In: *BMC bioinformatics* 13.1 (2012), p. 207.

[12] T. Lavergne, O. Cappé, and F. Yvon. "Practical Very Large Scale CRFs." In: *Proceedings the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*. Uppsala, Sweden: Association for Computational Linguistics, July 2010, pp. 504–513.

[13] A. K. McCallum. "{MALLET: A Machine Learning for Language Toolkit}." In: (2002).

[14] T. Kudo. "CRF++: Yet another CRF toolkit." In: *Software available at http://crfpp. sourceforge. net* (2005).

[15] N. Okazaki. *CRFsuite: a fast implementation of Conditional Random Fields (CRFs)*. 2007.

[16] C.-H. Wei, H.-Y. Kao, and Z. Lu. "GNormPlus: An Integrative Approach for Tagging Genes, Gene Families, and Protein Domains." In: *BioMed Research International* 2015 (2015).

[17] J. Lafferty, A. McCallum, and F. C. Pereira. "Conditional random fields: Probabilistic models for segmenting and labeling sequence data." In: (2001).

[18] X. He, R. S. Zemel, and M. Á. Carreira-Perpiñán. "Multiscale conditional random fields for image labeling." In: *Computer vision and pattern recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE computer society conference on*. Vol. 2. IEEE. 2004, pp. II–695.

[19] B. Settles. "Biomedical named entity recognition using conditional random fields and rich feature sets." In: *Proceedings of the International Joint Workshop on Natural Language Processing in Biomedicine and its Applications*. Association for Computational Linguistics. 2004, pp. 104–107.

[20] Y. LeCun and Y. Bengio. "Convolutional networks for images, speech, and time series." In: *The handbook of brain theory and neural networks* 3361.10 (1995).

[21] R. Johnson and T. Zhang. "Effective use of word order for text categorization with convolutional neural networks." In: *arXiv preprint arXiv:1412.1058* (2014).

[22] J. L. Elman. "Finding structure in time." In: *Cognitive science* 14.2 (1990), pp. 179–211.

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

[24] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio. "On the properties of neural machine translation: Encoder-decoder approaches." In: *arXiv preprint arXiv:1409.1259* (2014).

[25] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. "Empirical evaluation of gated recurrent neural networks on sequence modeling." In: *arXiv preprint arXiv:1412.3555* (2014).

[26] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai. "Class-based n-gram models of natural language." In: *Computational linguistics* 18.4 (1992), pp. 467–479.

[27] S. Miller, J. Guinness, and A. Zamanian. "Name Tagging with Word Clusters and Discriminative Training." In: *HLT-NAACL*. Vol. 4. 2004, pp. 337–342.

[28] T. Hofmann. "Probabilistic latent semantic indexing." In: *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM. 1999, pp. 50–57.

[29] D. M. Blei, A. Y. Ng, and M. I. Jordan. "Latent dirichlet allocation." In: *the Journal of machine Learning research* 3 (2003), pp. 993–1022.

[30] P. Kanerva, J. Kristofersson, and A. Holst. "Random indexing of text samples for latent semantic analysis." In: *Proceedings of the 22nd annual conference of the cognitive science society*. Vol. 1036. Citeseer. 2000.

[31] P. P. Kuksa and Y. Qi. "Semi-supervised Bio-named Entity Recognition with Word-Codebook Learning." In: *SDM*. SIAM. 2010, pp. 25–36.

[32] B. Tang, H. Cao, X. Wang, Q. Chen, and H. Xu. "Evaluating word representation features in biomedical named entity recognition tasks." In: *BioMed research international* 2014 (2014).

[33] C. dos Santos, V. Guimaraes, R. Niterói, and R. de Janeiro. "Boosting named entity recognition with neural character embeddings." In: *Proceedings of NEWS 2015 The Fifth Named Entities Workshop*. 2015, p. 25.

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

[35] J. Pennington, R. Socher, and C. D. Manning. "GloVe: Global Vectors for Word Representation." In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP 2014)*. 2014, pp. 1532–1543.

[36] B. Settles. "Active learning literature survey." In: *University of Wisconsin, Madison* 52.55-66 (2010), p. 11.

[37] F. Olsson. "A literature survey of active machine learning in the context of natural language processing." In: (2009).

[38] S. Bird and E. L. Klein. *E. 2009. Natural Language Processing with Python–Analyzing Text with the Natural Language Toolkit*. 2012.

[39] S. Loria. "TextBlob: Simplified Text Processing." In: *TextBlob. Np* (2014).

[40] T. De Smedt and W. Daelemans. "Pattern for python." In: *The Journal of Machine Learning Research* 13.1 (2012), pp. 2063–2067.

[41] R. Řehůřek and P. Sojka. "Software Framework for Topic Modelling with Large Corpora." English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. `http://is.muni.cz/publication/884893/en`. Valletta, Malta: ELRA, May 2010, pp. 45–50.

[42] P. Liang. "Semi-supervised learning for natural language." PhD thesis. Massachusetts Institute of Technology, 2005.

[43] F. Chollet. *keras*. `https://github.com/fchollet/keras`. 2015.

[44] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio. *Theano: new features and speed improvements*. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop. 2012.

[45] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. "Theano: a CPU and GPU Math Expression Compiler." In: *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation. Austin, TX, June 2010.

[46] Y.-F. Lin, T.-H. Tsai, W.-C. Chou, K.-P. Wu, T.-Y. Sung, and W.-L. Hsu. "A Maximum Entropy Approach to Biomedical Named Entity Recognition." In: *BIOKDD*. Citeseer. 2004, pp. 56–61.

[47] Z. GuoDong. "Recognizing names in biomedical texts using hidden Markov model and SVM plus sigmoid." In: *Proceedings of the International Joint Workshop on Natural Language Processing in Biomedicine and its Applications*. Association for Computational Linguistics. 2004, pp. 1–7.

[48] J. M. Cejuela, P. McQuilton, L. Ponting, S. J. Marygold, R. Stefancsik, G. H. Millburn, B. Rost, F. Consortium, et al. "tagtog: interactive and text-mining-assisted annotation of gene mentions in PLOS full-text articles." In: *Database* 2014 (2014), bau033.