Medical Concept Extraction

Tristan Naumann
MIT EECS
tjn@mit.edu

Samantha Ainsley
MIT EECS
ainsley@mit.edu

Salman Ahmad
MIT EECS
saahmad@mit.edu

ABSTRACT

This paper presents a system for extracting medical concepts from patient hospital records. Hospitals and medical practices are increasingly switching to electronic medical records. This trend has opened the doors to exciting applications for natural language processing in medicine and biology. One such application is concept extraction: identifying high-level semantic labels in a body of text. Concept extraction on medical records is useful in automating analysis tasks that would otherwise have to be done manually. For example, hospital administrators may want to tabulate what medications are being prescribed the most. Insurance companies may want to ensure that patients receive proper care and are billed accordingly for fraud detection and auditing purposes. Government agencies may be interested in exploring and better understanding public health trends. Automatic concept extraction systems help to streamline such critical yet laborious tasks.

In service of that end, we present an algorithm and system that is capable of classifying words in medical records across four different medically-relevant categories: (1) problems, (2) tests, (3) treatments, and (4) non-relevant terms. This is useful not only in granting end users the ability to analyze medical records, but also in supporting other higher-level natural language processing and learning systems. This paper describes the system's design, implementation, and performance when tested on set of 500 pre-annotated medical records from Boston-area hospitals.

1. INTRODUCTION

Concept extraction is the process of identifying relevant phrases of interest from unstructured text. In the medical field, concept extraction is exceptionally useful for quick retrieval of salient information from dense medical records, and it is the first step in modeling trends in patient medical histories. Concept extraction falls under the more general task of information extraction, which is a well-studied problem across many disciplines. Significant advances in information extraction over the past decade can be credited to increasingly robust machine learning systems–extending these learning methods to medical concept extraction is a far more recent trend.

In this work, we present a system for extracting problems, tests, and treatments from medical records at comparable performance (approximately 0.8 F1 score on average in testing) to existing methods. Our classification strategies prove successful in spite our relatively limited data set, which is comprised of hospital records from two Boston area facilities. Our portable open-source system is implemented in Python with easily configurable and extendable feature set and three classifier options.

An additional Web-based service also allows users to input single sentences for concept extraction on the fly. With great ease of access, this system helps to advance a novel and important application of natural language processing in the health care domain.

2. BACKGROUND

While medical concept extraction is a more recent trend, some previous work has been done in the area, most notably as part of the challenge whence our data originates. Better studied are the machine learning techniques which are leveraged by our work. In this section, we provide a brief survey of the relevant work and these machine learning algorithms.

2.1 Related Work

Although recent, concept extraction in medicine is an increasingly well-studied problem. One of the main driving forces behind such research is the annual i2b2/VA Workshop on Natural Language Processing Challenges for Clinical Records, which challenges participants to develop a system for "the extraction of medical concepts from patient reports." Torii et al. [10], for example, developed a system using the same training data as us for the workshop. Their concept extraction system was trained using a tagger specifically developed for the biology domain, and their work shows the difficulty in porting trained systems across clinical documents from various sources.

In a work independent of the i2b2 workshop, Meystre and Haug [7] developed a system for automatically retrieving medical problems from free-form text. Their system addresses one of the labels extracted from our system and references a pre-defined Problem List rather than training on data that is presumed to be a subset of all possible terms. Tackling an interesting related problem, Happe et al. [4] present a system for automatic concept extraction from *spoken* medical reports in French, with robust performance in testing in spite of sensitivities to ambiguous pronunciations and acronyms, which are common in automatic speech recognition systems.

2.2 Machine Learning

Our system addresses the particular challenge of assigning concept labels to terms in medical records, but the task of assigning labels from multiple classes over a set of entries is a classic machine learning problem. Supervised multi-class classification algorithms assign labels to input data entries using a model trained on a data set of pre-labeled observations. There are countless methods that support multi-class classification; but, for the purpose of this work, we employ three well-known strategies (general Support Vector Machines, linear Support Vector Machines, and Conditional Random Fields) and compare their respective accuracies given our data sets and chosen features (Section 4.2).

2.2.1 Support Vector Machines

Support vector machines (SVMs) aim to maximize the margin between decision hyperplanes, which separate training examples in feature space. Training examples are represented as points in feature space that are mapped to create independent classification regions with the largest possible gap between regions. In use, the SVM maps new examples into this space and assigns a label based on the classification region in which the example point resides.

Often problem sets are not linearly separable in feature space. To overcome this challenge, SVMs often map the original feature space into a higher-dimensional space–the specific mapping is defined by a kernel function, which is chosen for efficiency based on the specific classification problem. Separating hyperplanes, which are defined as vectors, can then be chosen as linear combinations of feature vectors in the training set.

We tested a non-linear radial basis kernel SVM as provided by the open-source library, libsvm [2].

2.2.2 Linear SVMs

As mentioned previously, we cannot always expect linearly separable training data. However, for large enough data sets, linear SVMs and SVMs with nonlinear mappings yield similar performance. By eliminating the need for a kernel mapping, training becomes much less costly through use of a linear support vector machine.

We additionally tested a strictly linear SVM using the open-source library, liblinear [3], which provides an L2-loss linear SVM solver.

2.2.3 Conditional Random Fields

Conditional random fields (CRFs), unlike SVMs, use a probabilistic modeling method. The strength of CRFs in the case of classification for NLP is that it can take neighboring samples (i.e. context) into account when handling individual observations. A CRF is a type of undirected graphical model, which models a conditional probability distribution between observations and output values, which are represented as nodes. Given a sequence of words, this graph takes the form of a chain analogous to a Hidden Markov model. The conditional dependency of each output label on a given input node is computed using a feature function in which features are given specific numerical weights and combine to determine the label's likelihood.

We tested our features using an open-source CRF library, CRFSuite [8], in addition to our SVMs.

3. CHALLENGES

There are several unique challenges which arise due to the the nature of the dataset and the approach of leveraging traditional machine learning classifiers for natural language.

3.1 High Dimensionality

In order to utilize traditional machine learning classifiers for natural language processing, it is necessary to generate linguistic features for each item that requires classification. The linguistic features which characterize each item often do not have a meaningful strict ordering (e.g. it is not meaningful to say "muffin" $>$ "bread"). Therefore, it is necessary to represent each feature without a strict ordering as a binary dimension reflecting the presence or absence of that feature.

Using this binary dimension representation results in a high dimensional space (e.g. using a word itself as a feature results in a number of dimensions equal to the size of the vocabulary of the training data). Further, the high dimensional space is exceptionally sparse since each item generally only satisfies one binary dimension for the feature. For non-asymptotic cases found in practice, this generally means that the feature space grows much faster than the amount of data and most of the theoretical guarantees of traditional machine learning algorithms do not hold.

Further, our best intuition often falls short in high dimensional spaces [5]. So it is difficult to generalize the utility of a feature to the entire data set even if it appears to be relevant for a handful of documents.

3.2 Biased Data

Many machine learning classifiers assume relatively balanced data with respect to the class labels available. However, in the task of concept extraction it is most often the case that there is a null concept class indicating that the token is part of no relevant concept which is overwhelmingly more abundant than the concepts. The bias of the class labels in the data presents a challenge, because a classifier optimized with respect to accuracy will be inclined to simply assign all instances to the null concept class since doing so will result in relatively high accuracy. Such classifiers will not, however, perform well with respect to more meaningful measures such as precision, recall, or F1-measure.

3.3 Amount of Data

As with many statistically-grounded projects, there is often only a limited amount of data available. Within the medical domain this holds particularly true due to the expertise (and subsequent cost) required for proper annotation, and the inability to easily merge data from multiple sources – either due to differing formats in which it is stored/accessed or various legal restraints in place to protect patients.

Such dearth of training data means that the amount of test data is often much larger than the amount of training data available. Consequently, it is even more difficult to obtain decent results if the models trained even moderately overfit the training data – a particular risk due to the sparse distribution of high dimensional items as discussed previously.

3.4 Data Usage Agreements

In addition to limited availability of data, it is often difficult to obtain data even once a relevant source is found. For "open" data sets such as those provided by the i2b2 National Center for Biomedical Computing, one must sign a data use agreement in order to maintain compliance with the U.S. Health Insurance and Accountability Act (HIPAA) privacy and security rules. The approval process for "open" data sets can still take weeks, while access to more controlled data sets can often take months and require security measures such as single physical machine access.

4. DESIGN

The most critical part of designing this system was properly identifying the correct linguistic features and transforming them into a representation suitable for the classification task. This section details both the classification task and those features we explored.

4.1 Data Flow & Classification Task

Our system considers each document to be a stream of words grouped by sentences, and it attempts to provide the correct concept classification for each word (i.e. it performs classification at the token level). In order to identify the appropriate concept, our system considers natural language features for each document in successively broader contexts. Specifically, we first consider features using only the word itself as context, next we consider features which span the sentence, and last we consider n-gram features. In the sections that follow we discuss the linguistic features used in each context.

4.2 Word Features

Using only the word as context we are able to generate several interesting linguistic features. These features generally take on one of three forms: (1) intrinsic property of the word, (2) binary indicator for properties that we think are interesting, and (3) various techniques that map words into a small dimensional space such as stemming and word shape.

4.2.1 Word

The most obvious property of each word is perhaps the word itself. Since there is no linguistically meaningful, strict ordering over words, each word is represented as its own dimension. Consequently adding a baseline feature containing the word increases the dimension of our feature space to the size of the vocabulary in the training data. Nevertheless, the complete word serves as a fruitful feature in medical term classification in which many words carry the same meaning regardless of context. For example, terms such as "x-ray", "screen","exam", "veal", and "sats" should be consistently recognized as test terms, just as "swelling", "failure", "trauma", and "syndrome" always indicate problems, and "therapy", "anesthesia", "vaccine", and "medication" are treatments. Granted, words like "heart" could be part of "heart rate" or "heart attack" and beg for more generic features.

4.2.2 Length

The next most obvious property of each word is its length. Unlike using the word as a feature, there is a strict ordering over length and consequently the addition of this feature increases the dimension of the space by only 1. While the linguistic merit of length as a feature can be debated in conventional natural language processing, we felt it would play a large role in medical language in order to distinguish treatments, tests, and problems from words without a concept. Intuitively this is because often times treatments in the form of medications consist of long medical names, as do some of the words in medically described problems and tests.

4.2.3 Regular Expressions

In order to capture some general characteristics of each word we included a set of regular expressions, each of which adds a dimension to our classification. Such features help allow us to pick up on traits within different medical categories–such as tests which are often abbreviated or contain numbers–and to easily rule out non-medical terms such as days, phone numbers, etc. The characteristics we chose are as follows:

- `INITCAP`: first letter is capitalized and alphabetic,
- `ALLCAPS`: entire word is capitalized and alphabetic,
- `CAPSMIX`: the word is a mix of capital and lowercase alphabetic characters,
- `HASDIGIT`: there is a digit in the word,

- `SINGLEDIGIT`: the token is a single digit,
- `DOUBLEDIGIT`: the token is a two digits,
- `FOURDIGITS`: the token is a series of four digits,
- `NATURALNUM`: the token is a natural number,
- `REALNUM`: the token is a real number,
- `ALPHANUM`: the token contains only digits and alphabetic characters,
- `HASDASH`: the token has a dash,
- `PUNCTUATION`: the token is punctuation,
- `PHONE1`: the token is a phone number,
- `PHONE2`: the token is a phone number with area code,
- `FIVEDIGIT`: the token is a series of five digits,
- `NOVOWELS`: the token does not contain vowels (i.e. abbreviations),
- `HASDASHNUMALPHA`: the token consists of a dash, numbers, and alphabetic characters,
- `DATESEPERATOR`: the token is a date separator (e.g. `[-/]`)

4.2.4 Porter Stemmer

We already discussed the benefits of using entire words as features, but similar words come in many different forms. Motivated by generalization, we chose to use word stems as features and approached this feature extraction using three different stemming algorithms–two at the word-level and one at the sentence level (See 5.3.2). The addition of word stem features to our system improved our toy test CRF F1 scores from 0.58 to 0.82. We used built-in NLTK implementations of all of the stemming algorithms described in future sections.

Namely, we used the Porter Stemming Algorithm, as developed by Martin Porter. This particular stemming algorithm is divided into a number of linear steps. Word letters are first replaced by symbols for vowels, V, and consonants, C. The measure, m, of the word is computed as the number of repetitions of the pattern VC, and this is used to determine whether suffixes ought to be removed. Plural word forms are then removed such as $sses \rightarrow ss$ and removal of trailing s's; followed by the removal of past participles; and finally terminal y's are replaced with i's. Subsequent steps handle different orderings of suffixes by translating double suffixes into a single suffix and then removing suffixes under certain predefined conditions.

4.2.5 Lancaster Stemmer

Derived from the Porter Stemmer, the Lancaster Stemmer was used to provide a second word stem feature to our system. By constant, this stemming algorithm uses a table of rules that specify the removal or replacement of an ending. Endings are replaced to maintain properly-spelled words throughout the stemming process–this avoids the need for an additional re-coding or partial matching phase.

The rule table looks for the following information about each word: Endings of one or more characters (stored in reverse order); an optional intact flag; a digit specifying the removal total; an optional append string of one or more characters; and a continuation symbol. The stemmer first inspects the final letter of a term and considers the first relevant rule in the table. If the rule conditions are not satisfied, another rule is considered, otherwise the rule is applied and the ending is removed or reformed. This process is repeated until a termination symbol is uncovered and the word is presumed to be in its stem form.

4.2.6 Word Shape

In the same spirit of our word length and MIT RegEx features, we decided to look at the high-level shapes of individual words for better generalization. More specifically, we encode words into new strings that give a sense of their capitalization and punctuation patterns as well as approximate length. We tested a variety of open-sourced word shape classification methods including two methods written by Dan Klein [6], one of which simply classifies words as numeric, strictly upper or lower cased, or mixed, as well as a second, more fine-grained classifier that attempts to group words into equivalence classes by collapsing sequences of the same type. Additional classifiers developed by Christopher Manning [6] seek to make less distinctions mid-sequence by sorting words based on length. Each word shape classifier result added an many more dimensions to our feature space. By naively including all word shape alternatives as features, we were able to improve our toy test CRF F1 score from 0.82 to 0.85.

4.2.7 Additional Feature Considerations

The word-level features that we ultimately decided to include in system were chosen based on performance gain, but many others were explored in testing. These features were selected after close examination of our training data, and are worthy of discussion. It is possible, of course, that there is a more optimal feature set than our final choice that includes the features listed below:

- `Metric Units` A clear trend we noticed in our medical record data was that numbers followed by metric weights (i.e. mg, milligrams, grams) are almost always part of treatment descriptions. Whereas numbers followed by metric lengths or areas are typically part of problem observations. We still believe this could be a useful feature for related systems.

- `Word Endings Indicative of Medical Problem Type` Another obvious trend specific to medical problems are diseases ending in "itis" such as "bronchitis", "cellulites", "fibrositis", etc. and conditions ending in "ic" such as "diabetic", "anemic", etc. We added a regular expression to flag words with these endings, but the feature did little to nothing to improve performance. Such observations were likely already handled by our other word features.

- `Prognosis Locations` Many diagnoses include locational information that should be tagged as part of the problem (i.e. "C4-C5 herniated disks"). We still think that a robust feature of this form could benefit our system, but our particular attempts proved unsuccessful.

- `Flag Words` Certain terms such as those described in 5.2.1 are consistently labeled as "problems", "tests", or "treatments". We attempted to create small dictionaries of these terms and assign flags to words that fall into one of the three dictionaries. The issue with this strategy is that words that should be labeled as one of our three medical terms but do not appear in the dictionary are now more closely aligned with purely non-medical words. That said, this feature improved accuracy for our three categories, but also yielded far too many false positives.

4.3 Sentence Features

Even though the primary goal of our system is to tag concepts at the word level, these individual words often need to be considered in context. After extracting the word-level features described in 5.2, we then analyze medical records at the sentence level and assign features to our words based on their usage within those sentences.

4.3.1 Part of Speech

A fruitful strategy for any concept extractor is to first tag words with labels indicating their part of speech. In our case, for example, whether an ambiguous term is used as an adjective versus a noun could distinguish it as part of a test rather than a problem. We used the NTLK max entropy part of speech tagger, which is trained on the Penn TreeBank to tag each sentences. We then iterate over the returned tags and add these as word-level features, adding several dimensions to our feature set, one for each part of speech tag.

4.3.2 WordNet Lemmatizer

Unlike the Porter Stemmer and the Lancaster Stemmer, which consider words independent of context, the WordNet Lemmatizer requires words to be processed at the sentence level in order to extract their root form. The WordNet Lemmatizer uses the part of speech tags for individual words, which are assign via NLTK as previously mentioned, in conjunction with a subset of tags used for morphology to extract stems from each word. WordNet references a database of word stems, and uses two types of processes to try to convert input strings into a form present in the database: A list of inflectional endings based on the word's syntactic tag is referenced in search of a match that can be detached from the word. An exception list is also referenced in search for an inflected form. WordNet first checks for exceptions and then uses rules for detachment to transform the word. After each transformation the result is referenced against the stem database until a match is eventually found.

4.3.3 Formatted Text

When reading through our training data, we noticed that test results are written in a variety of distinct forms that clearly separate these terms from problem or treatment terms. Since test result forms often spanned multiple words such was "O2 sat – 98%" or "screen was positive". We wrote a series of regular expressions to capture these patterns with the test term as the first term in the sequence. We iterate over each word in the sentence and run our RegEx suite on the inclusive righthand sequence starting from that word. This feature adds only one additional dimension, and improved our toy test CRF F1 test score from .85 to .90 by boosting our test term detection appreciably.

4.4 N-gram Features

Again, although our primary goal is to classify individual words, a single word in a medical record entry is hardly independent from the next. More specifically, medical concepts can often span more than one word. We may not be interested in detecting these exact spans, but features from certain words provide insight into how to label their neighbors. For example, ambiguous words such as "heart" could precede "rate" or "disease", and making that distinction between test and problem is much easier when we examine the subsequent word in the sentence containing "heart". A neighboring word can serve as an indicator even if it is not part of the same concept span. For example, words like "showed" and its synonyms are non-medical, but tend to follow a test term. Likewise, "rule out" and "history of" tend to precede problems. With this in mind, we observe all features of immediately neighboring words.

4.4.1 Previous

We take a basic approach to context features by considering the immediately preceding word for each individual word. Given a word with the feature set previously described, we simply duplicate each feature with the corresponding value for it's previous neighbor and rename these features *prev_*<NAME> (e.g. *prev_*length). This serves to double the size of our feature space.

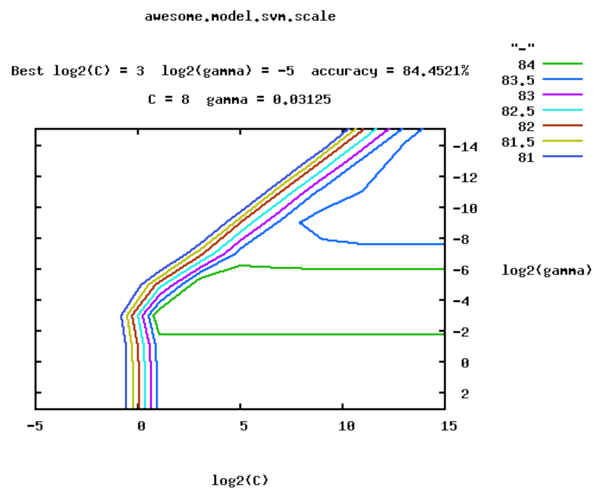


Figure 1: An illustration of the grid search process to identify the best parameters values for training the SVM model.

4.4.2 *Next*

Likewise, we duplicate the subsequent word features, which are relabeled as *next*_<NAME>. At this point, our feature space is tripled, but was already quite large–hence, we decided only to consider surrounding words at in this "bigram"-like context. Nevertheless, this relatively straightforward step brought our toy test CRF F1 testing score up to 0.98.

4.5 Parameter Tuning

Due to the data's bias with respect to labels which we previously discussed, the default SVM parameters failed to provide any sort of relevant results. Therefore, with our full feature set we used libsvm's included utilities to perform parameter selection as shown in Figure 1. The utility performed 5-fold cross validation and allowed us to see the parameters which provided the best accuracy. In turn, this enabled us to find an appropriate constant C with which to scale the loss function and constant γ for the radial basis function kernel the SVM was using.

It should be noted that the best method would be to optimize against the F1 score rather than the accuracy due to the data bias previously discussed. However, in practice, the parameters that were discovered using this method were able to dramatically improve performance while still being easy to obtain.

5. SYSTEM

The system we have developed consists of a framework in order to perform concept extraction, a Web service frontend which allows users to test the service without seeing any of the underlying data, and the machine learning libraries that are leveraged.

5.1 Code Architecture

We built a fairly comprehensive framework for applying machine learning techniques to NLP domains. We built out a series of useful and reusable library modules and command line utilities to perform various tasks. An overview of the codebase can be seen in Figure 2 and all of the code is available online at the following URL:

`https://github.com/tnaumann/ConceptExtraction`.

5.1.1 *Data Formats and File I/O*

Our framework consists of many convenience functions and utility classes that are designed to perform all necessary file I/O operations–formatting output data and parsing input files. There are 3 formats that the code base handles.

The first format is a straight forward tokenized text file that contains each sentence on a separate line. The test data for training the models were processed into this format. For cases where the sentence was not previously tokenized, the framework utilizes `NLTK`'s `sent_tokenize` to break a string into sentences and then `NLTK`'s `word_tokenize` to break each sentence into normalized tokens.

The second format is an annotation format that references the tokenized text files and associates a classification label ("problem", "tests", "treatment" or "none") with spans of words. This format is used in a separate file than the actual input tokenized text file but the two files typically share the same name so we can tell which labels apply to which input. The training and test data sets use this format to represent the ground truth classification of the medical records.

The third and last format is is a sparse matrix format that is utilized by the various command-line utilizes in the code base. Conceptually, the rows in this sparse matrix represent an a single data item (in our case, a single word) that either is a training example or is an input token that needs to a label predicted and the columns represent the various features of that item. Each of the various learning algorithms use this data format.

5.1.2 *Model Encapsulation and Serialization*

Our framework includes a model class that can be trained on specified data with any combination of features enabled. The model class supports training a libsvm, liblinear, and crfsuite classifiers that can be used for prediction in the future. One of the nice things, about the model class is that it can be serialized and reloaded dynamically. This is a really nice capability because it allows you to train a large number of models on various subsets of the data and using various different features to see which perform the best.

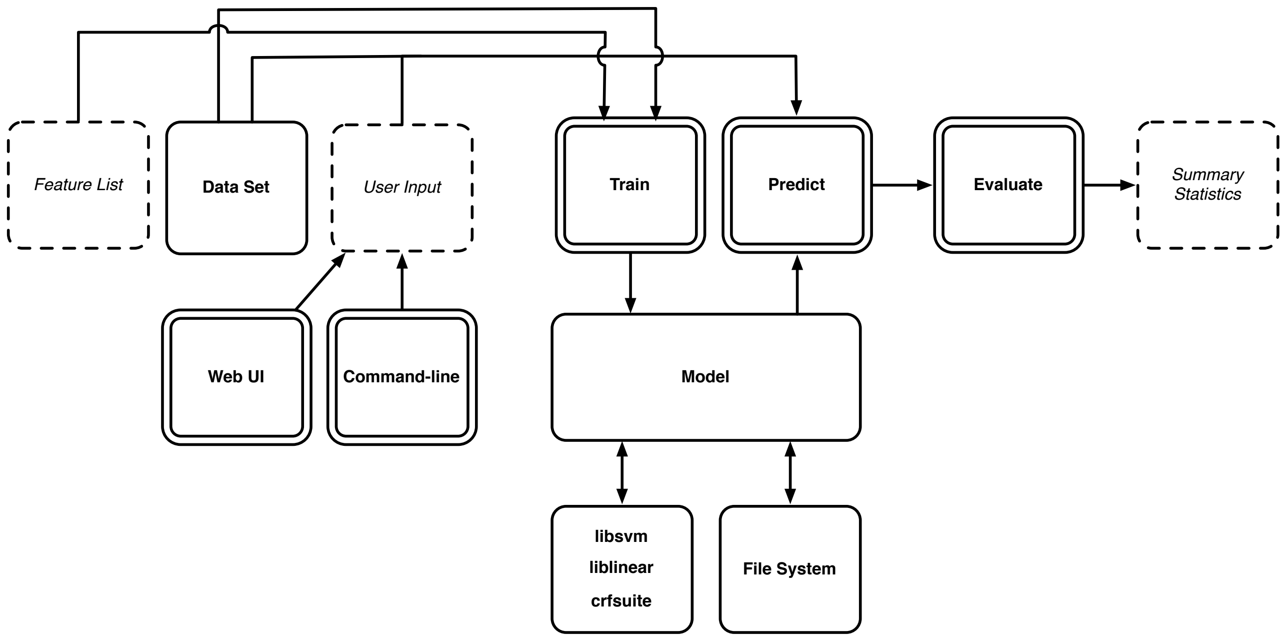


Figure 2: An overview of the system and codebase.

5.1.3 Feature Computation

The framework has a flexible and robust mechanism for computing features on input tokens. Features are implemented as a simple conditional statement in a large `if-else` statement, making it really easy to add new features to the system. Additionally, each model maintains a list of the features that are enabled. Only the enabled features will be executed and computed on the input tokens. Features can be enabled or disabled pragmatically by removing them from the model's feature list. This ability is really important when training models because it enables you to easily explore the impact of certain features and run feature selection.

5.1.4 Learning Algorithms

As mentioned previously, the framework's model class has support for training various classifiers: a Support Vector Machine, a CRF, and L2-regularized Support Vector Machine. The implementation of these algorithms is delegated to various third-party libraries described at the end of this section.

5.1.5 Data Organization and Directory Structure

The framework supports a default conventional directory structure to know where to store test, training, and prediction data. This simplifies training new models and testing how well they perform. While these directories can be configured, the defaults are:

- `data` - Contains all of the data that is used for training and large testing runs. Again, this is just conventional and it is possible to train and predict on data that is not in this directory. For example, we are able to predict "one-off-sentences" as a command-line argument.

- `data/<data-set-name>/txt` - Contains the raw tokenized text files.

- `data/<data-set-name>/concept` - Contains the labels that correspond to each of the tokenized files in the "txt" directory.

- `models` - Contains all of the trained models.

- `models/<model-name>` - Contains the meta data specific to each model that was trained. The framework saves the model's internal state (the features used, the input test set, etc.) as well as the trained parameters for the libsvm, crfsuite, and liblinear classifiers. This information is used to deserialize and load a model at runtime for prediction tasks.

- `models/<model-name>/test_predictions/` - Contains output files with the predicted label for each word in the input file.

5.1.6 Command-line Utilities

The framework contains three useful command line utilities to interact with the system. The functionality provided by these utilities are also exposed by a Python API so other developers can interact with the system natively in their own Python scripts without having to fork calls to the command line.

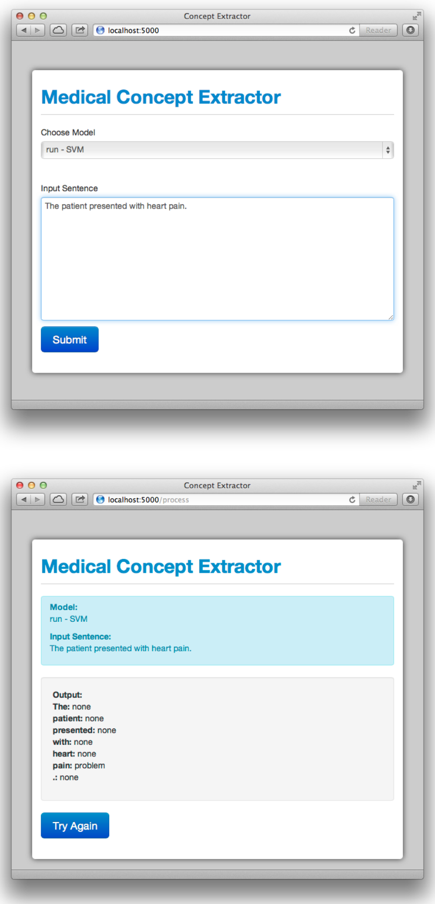
- `train.py` - Trains a model using the specified training data and output model name.
- `predict.py` - Predict the labels for the specified input files and write the labels to a designated file.
- `evaluate.py` - Compare the predicted labels to the gold standard labels and print outs the confusion matrix, precision, recall, and F1-scores.

5.2 Web Service

In addition to the various command line utilizes described in the previous section, our system also includes a Web-based service that allows users to manually input a single sentence, select a model to use for prediction, and see the predicted labels that model generated for the input. Figure 3 shows a screenshot of the Web application in use.

The Web service was written in Python and directly interoperates with the rest of the code base. To run the Web applications simply execute the `web.py` file in the source tree. The Web app allows you to label a sentence using any of the models that we have previously trained. This is useful for spot checking certain sentences and observing how different models behave. The Web app assumes that models have already been trained and placed into "models" directory in the source tree. By default, the application will host itself on port 5000. You can also see a live version of the Web service at:

`http://saahmad.media.mit.edu:5000/`


Concept Extractor
localhost:5000

Medical Concept Extractor

Choose Model
run - SVM

Input Sentence
The patient presented with heart pain.

Concept Extractor
localhost:5000/process

Medical Concept Extractor

Model:
run - SVM

Input Sentence:
The patient presented with heart pain.

Output:
The: none
patient: none
presented: none
with: none
heart: none
pain: problem
.: none

Figure 3: The Web application interface that allows users to manually input a sentence and have it labeled using one of the previously-trained models. In this case we are using an SVM model that was trained using all of the features.

5.3 Libraries

Our code leveraged various 3rd party libraries for training the models and generating features. A summary of the libraries that we used is shown in the list below:

- `libsvm` [2]: libsvm provides a fast implementation of Support Vector Machines. It includes a command line utility that takes a training data (with all of the features formatted in a sparse matrix representation) as input and outputs a SVM model trained on that data. It also includes a command line utility that allows users to specify a desired model and predict the classes on new input. Our code base forks process calls to these utilities to train SVM models, although a lot of the parameter tuning and feature selection is handled elsewhere in our codebase.
- `liblinear` [3]: liblinear is a linear classifier library that has special support multi-class classification. It also provides easy to use training and predicting command line utilities that accept a similar input format as libsvm.
- `crfsuite` [8]: crfsuite is an implementation for conditional random fields and includes command line front ends to train a model on data and to predict the labels on an unseen data set. The data format for the input training files is also a sparse matrix representation but is slightly different from the format used by libsvm and liblinear. Our codebase nicely abstracts out these format differences and produces the right format depending on the model in use.
- `NLTK` [1]: NLTK is a very popular NLP library written in Python. We leverage NLTK to generate some of the features on our data. In particular, we utilize NLTK's maximum-entropy part-of-speech tagging as well as many of its stemming algorithms.
- `Flask` [9]: Flask is a micro Web framework written in Python. It contains a small yet useful set of features that allow developers to easily setup RESTful Web backends and an elegant templating system to gracefully generate HTML. Flask is used to build the Web service portion of the project.

Training Data Set		
Label	**Count**	**Percentage**
test	8492	5.7
none	114783	76.7
problem	17326	11.6
treatment	8940	6.0
Total	149541	100

Testing Data Set		
Label	**Count**	**Percentage**
test	17237	6.4
none	202438	75.8
problem	30276	11.3
treatment	17298	6.5
Total	267249	100

Figure 4: The number of each class the in testing and training data sets.

6. RESULTS

This section discusses the data set, the models that were trained as part of the project, the (excessive) amount of time taken to train them, and the performance of those models on the test data.

6.1 Data Set

Deidentified clinical records used in this research were provided by the i2b2 National Center for Biomedical Computing funded by U54LM008748 and were originally prepared for the Shared Tasks for Challenges in NLP for Clinical Data organized by Dr. Ozlem Uzuner, i2b2 and SUNY. As mentioned before, the data came divided into training and test sets and every word was pre-labeled with the following classes: "problem", "test", "treatment", or "none".

The overall size of the data was 10mb and came from Boston-area hospitals–so it is subject to data use agreements. The training data consisted of 170 medical records and the test data consisted of 256 medical records. Further, the distribution over labels of the training and test sets is provided in Figure 4.

6.2 Trained Models

Overall we trained a total of 23 models using the entire data set. One model was trained with all of the features. We then trained a set of 11 models in which we removed a single feature as well as 11 models in which we had only one feature enabled. Even though this is not as complete as a running full feature selection, it will hopefully hint at the most useful features.

Recall that each model contains three different classifiers: libsvm, liblinear, and crfsuite. So, in total we attempted training 69 classifiers. Unfortunately, as discussed later in the paper, the libsvm classifiers took very long to train and most were not finished in time. Consequently, their results are not included in this report. Fortunately we were able to train a single libsvm classifier from a model that had all of the features enabled. So, all in all the results include the output from 47 classifiers (46 liblinear and crfsuite classifiers and 1 libsvm classifier).

crfsuite					
		Predicted			
		test	problem	none	treatment
Actual	test	14388	1503	932	414
	problem	288	27993	1521	474
	none	3214	18463	175032	5729
	treatment	304	1896	1418	13680

liblinear					
		Predicted			
		test	problem	none	treatment
Actual	test	11819	1171	3718	529
	problem	475	22682	6383	736
	none	1120	4408	195085	1825
	treatment	249	1251	4099	11699

libsvm					
		Predicted			
		test	problem	none	treatment
Actual	test	12741	1179	2935	382
	problem	542	23288	5818	628
	none	1284	4175	195716	1263
	treatment	438	1989	3860	11011

Figure 5: Confusion matrix for a model trained on all of the test data with all of the features enabled.

6.3 Runtime Performance

We deployed the system on a machine running Ubuntu 11.10 with a QuadCore AMD Opteron processor running at 2.5Ghz and 48GB of RAM.

In general training the liblinear and crfsuite classifiers was pretty quick. Each took around an hour to train on the entire training set. However, the libsvm classifier took approximate 6 hours to train on the same data.

In terms of predicting labels for unseen input, all classifiers performed equally well. They each took around 1 hour to label the entire test data set. However, the vast major of time was spent computing features on the input rather than actually running the classifiers.

6.4 Classification Performance

Overall we found that our system did a good job at predicting labels for medical notes. Figure 5 shows a confusion matrix of a model that was trained with all features enabled and Figure 6 shows summary statistics for that very same model. The results are broken down so we can see the relative performance of each of the individual classifiers. Overall, the system does very well and achieves an average F1-score of about **0.80**.

Feature Set											F1 Score	
length	mitre	next	pos	prev	stem_lancaster	stem_porter	stem_wordnet	test_result	word	word_shape	LIN	CRF
✓											0.2157	0.0420
	✓										0.2155	0.0927
		✓									0.2155	0.0572
			✓								0.2155	0.0820
				✓							0.2155	0.0366
					✓						0.7085	0.2878
						✓					0.7266	0.2886
							✓				0.7309	0.2857
								✓			0.2806	0.1084
									✓		0.7332	0.2753
										✓	0.2431	0.1244
	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	0.7986	0.5851
✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	0.7911	0.5937
✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	0.8049	0.6077
✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	0.8101	0.5851
✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	0.8101	0.5883
✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	0.8033	0.5818
✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	0.8004	0.5163
✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	0.8022	0.6442
✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	0.7487	0.4603
✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	0.7962	0.6534
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		0.7981	0.5705
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	0.8001	0.7891

Figure 7: A summary of the F1-scores from many different models with different features enabled.

As mentioned before, we trained a total of 47 classifiers with different features to get an idea for what features were the most important. A summary of the F1-scores of these classifiers is show in Figure 7.

7. DISCUSSION

As previously mentioned, we tested our feature set using three classifiers in search of the best concept extraction system (general SVM, linear SVM, and CRF), all of which yielded comparable results in terms of their average F1 scores, but showed significant difference in terms of efficiency. Our least time efficient classifier, SVM, was eventually eliminated from our system, as discussed in the subsequent section.

After extensive testing of various subsets from our initial feature set, the most optimal combination we discovered was the inclusion of all features discussed in Section 4.2 less the previous word n-gram feature and the part of speech tags using a general SVM. The previous word feature was likely redundant in conjunction with the next word feature; likewise, the part of speech tags are reused in the WordNet Lemmatizer word stem feature. Certain features (metric units, word endings, prognosis locations, and flag words as discussed in Section 4.2.7) were eliminated before significant testing was performed as they proved to have an appreciable negative impact on our F1 scores when testing toy examples.

Our more extensive tests also revealed that our baseline F1 score is primarily a reflection of the inclusion of a single feature: the word itself. In that sense, much of our performance can be credited purely to machine learning with one particularly strong feature. However, our additional improvements on this score required more in-depth linguistic analysis of our training data. Moreover, all of our machine learning algorithms showed comparable improvements with more language-driven features. Therefore, these features indicate a measure of success that is independent from the chosen canned classifier. Nevertheless, we can always argue for a more eloquent implementation of our system in spite of our successful results.

8. CONCLUSION AND FUTURE WORK

While the libsvm-trained SVM was able to perform the best overall classification performance, the exorbitant time required to train makes it of little practical use. Both the liblinear-trained L2 regularized SVM and crfsuite-trained CRF provide comparable classification performance in extremely reduced training time, and roughly equal prediction time. In fact, the runtime difference was so great that we were able to begin enumerating the feature space with both

crfsuite			
Label	**Precision**	**Recall**	**F1**
test	0.8650	0.6857	0.7650
problem	0.7686	0.7492	0.7587
none	0.9321	0.9637	0.9477
treatment	0.7911	0.6763	0.7292
Average:	0.7512	0.8537	0.7891

liblinear			
Label	**Precision**	**Recall**	**F1**
test	0.7908	0.8347	0.8122
problem	0.5615	0.9246	0.6987
none	0.9784	0.8646	0.9180
treatment	0.6740	0.7908	0.7278
Average:	0.8392	0.7687	0.8001

libsvm			
Label	**Precision**	**Recall**	**F1**
test	0.8491	0.7392	0.7903
problem	0.7603	0.7692	0.7647
none	0.9395	0.9668	0.9529
treatment	0.8289	0.6365	0.7201
Averages:	0.8444	0.7779	0.8070

Figure 6: Results from a model trained on all of the test data with all of the features enabled.

the liblinear and crfsuite models well before the first libsvm classifier was trained.

Across the board, we found F1 scores above what we had expected. And thus we believe this method of identifying linguistic features and then leveraging statistical techniques is successful. While we were able to develop several strong models, there were several additional ideas that piqued our interest which we must defer to future work for the sake of time.

8.1 Dimension Collapse

There are several mechanisms we thought of to to collapse the high dimensional nature of the data. Given unbounded time, we would have liked to try the following.

- First, we used stemming to add additional features to each word much in the same way that the word feature was created. However, after performing more complete feature selection we would have liked to select the best suited stemmer and use that in place of the word feature.

- Depending on the stemmer selected, we may have also added a list of stopwords that mapped to a single dimension in order to gain a dimension saving over "uninteresting" words.

- Next, we would have liked to investigate a dramatic reduction by creating a list of all words that are only tagged with the null concept class and mapped them all to a single dimension.

While the second approach exhibits only a modest reduction (and may be taken care of automatically by some stemmers), the first and third approaches may allow us to dramatically reduce the dimension of the problem. Our hypothesis is that this will certainly improve the runtime performance, and perhaps the classification performance and model generalization as well.

8.2 Feature Selection

In our work, we performed feature selection in-so-far as we thought of a large number of potential features and then tested their efficacy on a small subset of documents. Further, we generated models from the full set of features as well as each individual feature. While these proved effective for the purposes of analysis, a more exhaustive feature selection is certainly desirable.

8.3 Parameter Selection

Likewise, we experimented with parameter selection for the SVM classifiers over a reduced data set and were able to obtain dramatic increases in classification performance. Due to the already long runtime of training SVM classifiers, it was not feasible to perform any sort of robust parameter selection for these models. However, we expect that doing so would provide additional gains.

8.4 More Data

It goes without saying that having more data is always nice. This is particularly true in our case because we were training on a relatively small data set and testing on a considerably larger one. In fact, an interesting avenue for future work is to switch our training and testing data sets so we are training on relatively larger data set.

Additionally, the training data is derived from clinical notes originating at Beth Israel Deaconess and Partners Healthcare. It would be interesting to test our system against data from health centers outside the Boston area which have differently organized notes to see if our system is able to generalize well. Because many of our features avoided encoding language structure that appeared in the training data when possible, we expect this generalization would be relatively good assuming that the vocabulary used in an alternate test set was similar to our training set. However, this could very easily not be the case.

9. ACKNOWLEDGMENTS

We sincerely thank Dr. Robert Berwick and Geza Kovacs for their guidance, help, and support. May the Force be with you, always.

10. REFERENCES

[1] S. Bird, E. Klein, and E. Loper. *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly, Beijing, 2009.

[2] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[3] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[4] A. Happe, B. Pouliquen, A. Burgun, M. Cuggia, and P. L. Beux. Automatic concept extraction from spoken medical reports. *International Journal of Medical Informatics*, 70:255–263, 2003.

[5] B. Hayes. An adventure in the nth dimension. *American Scientist*, 99(6):442–446, 2011.

[6] D. Klein and C. Manning. Stanford nlp library: Wordshape classifier.

[7] S. M. Meystre and P. J. Haug. Comparing natural language processing tools to extract medical problems from narrative text. *AMIA Annu Symp Proc.*, pages 525–529, 2005.

[8] N. Okazaki. Crfsuite: a fast implementation of conditional random fields (crfs), 2007.

[9] A. Ronacher. Flask: A microframework for python based on werkzeug and jinja2, 2012.

[10] M. Torii, K. Wagholikar, and H. Liu. Using machine learning for concept extraction on clinical documents from multiple data sources. *J Am Med Inform Assoc*, 18:580–587, 2011.