Extending WordNet using Generalized Automated Relationship Induction

Lawrence McAfee lcmcafee@stanford.edu
Nuwan I. Senaratna nuwans@cs.stanford.edu
Todd Sullivan tsullivn@stanford.edu

This paper describes a Java package for automatically extending WordNet and other semantic lexicons. Extending these semantic lexicons by traditional means of hand labeling word relationships is a very expensive and laborious process. We used machine learning techniques to automatically extract relationships between words from a given text corpus. The package is made to be very flexible, allowing for various modules, such as new classifiers and semanitic lexions, to be "plugged-in." The power of the package comes form its ability to seamlessly integrate the Stanford Parser to WordNet. Results obtained for various tests, particularly those done for a Naïve Bayes classifier, are promising.

1 Introduction

WordNet-like semantic lexicons are vital for research and application development in natural language processing and related fields. However, attempts at extending such lexicons using traditional methods such as hand-crafted patterns or human insertion of new words into the taxonomy to cover more general vocabularies have proved to be very expensive and tedious. Recently, there has been some work into using machine learning-based methods to automatically learn word relationships from large collections of text.

The goal of our project is to use machine learning techniques to build a framework that is capable of identifying a range of word relationships. Our package is general enough to be able to handle most types of relationships that can be observed within single sentences. The large amount of flexibility in our package is due to its extensibility characteristics, allowing the user to add their own custom classes to certain packages.

Our package is also very powerful because of its ability to integrate the Stanford Parser and WordNet into a single package. The Stanford Parser is used to parse sentences into tree structures, from which we can extract patterns that are used in our feature vectors.

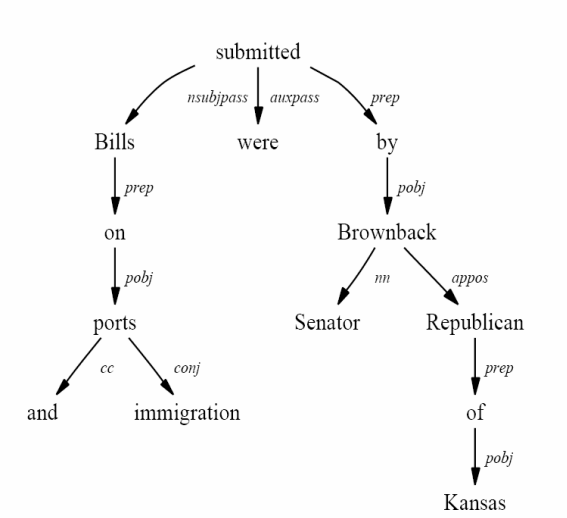

Figure 1: Example of a typed dependency tree generated by the Stanford Parser [2]

Previously, an algorithm was developed by Snow et al. to automatically induce new word pairs from text that exhibit the hypernym relationship [1]. We build upon work in [1] by creating a generic word induction algorithm that can operate on any word relationship where the word pairs exist in the same sentence. Additionally, we invoke the induction process described in [1] repeatedly, training on the previous training set and the new word pairs generated from the previous iteration. Snow showed that a good technique for the hypernym relationship (as is also for our algorithm) to capture structured relational knowledge is to use dependency trees. Dependency trees represent dependencies between individual words in a sentence.

Our project improves on the general layout of this system by 1) increasing accuracy through the use of a better parser (the Stanford Parser), 2) broadening the use for such an algorithm by incorporating a wider range of word relationships, and 3) by being able to train our algorithm on a much larger corpora than was previously available. Our package will make use of the Stanford

Parser (for creating sentence tree structures from text), WordNet (for supplying word relationship example data) among other semantic lexicons, and various classifier packages to compare and determine what works best for word-pair induction, including SVMs, Naïve Bayes, and logistic regression, among others.

2 Design

The design layout of the Generalized Automated Relationship Inductor (GARI – see Figure 2) is based on the following process:

Given a general semantic relationship, an initial training set of both known-to-be-related and known-not-to-be-related word pairs for a given relationship and a large corpus of text, GARI will use the training pairs to discover patterns in sentence tree structures derived from the corpus that indicate the relationship. The two tree structures we tested were the context-free phrase structure trees (also known as Penn trees) and typed dependency trees.

Once such patterns are discovered, a classifier is trained to predict whether an arbitrary pair of words occurring in the corpus is related by the given relationship. This process will allow the discovery of yet unknown instance pairs of the relationship.

The newly discovered (or "induced") pairs will then be added to the training set and the above process will be repeated in a "chain reaction" fashion (see section Feedback Loop). The process continues until we observe a reduction in the quality of the induced results.

2.1 Component Libraries

We have designed GARI as a set of component Java Libraries. These component libraries may either be used separately or combined as a whole. Our design consists of four principal packages: `stanfordparser`, `relationshipverifier`, `feature`, and `classifier`. The packages are combined to provide GARI's core functionality. Alternatively, if the user wishes to exploit them individually, they may be used separately.

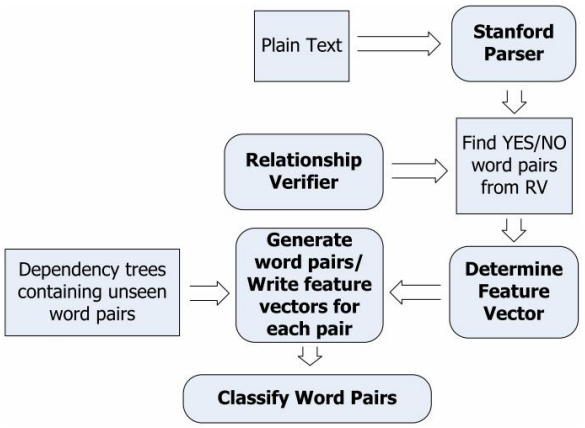

Figure 2: Layout of GARI

2.2 Package `stanfordparser`

Given a corpus of plain text, we use the Stanford Parser to generate a corresponding set of either Penn trees or typed dependency trees. We then find the dependency path between pairs of relevant words for tree (for example, the path connecting two nouns in the case of the hypernym relationship). This package interacts with the Stanford Parser Java API.

2.3 Package `relationshipverifier`

This package interfaces GARI with sources that provide the initial training data needed to initiate the induction process. It contains functionality to interface GARI with WordNet, as well as interfaces that would enable GARI to obtain initial training data from alternative semantic lexicons, and other data sources including plain text files. The interface with WordNet interacts with the WordNet Java API. The WordNet interface generates pairs of words from WordNet that are known-to-have and known-not-to-have a given relation. This set of word pairs is pruned to those that exist in the corpus, and the resulting set is saved to disk for later use.

This is a flexible part of the system, as it allows the user to "plug-in" any semantic lexicon that contains example word pairs for a given relationship. Given the relation specified by the user, any word relationship contained in a single sentence can be learned. We have, as default functionality, supported relationships of antonyms, holonyms, hyper/hyponyms, meronyms, participles, and synonyms.

2.4 Package `feature`

Once we have converted our text corpus into tree structures and we have obtained the initial sets of training data, we then extract relevant sub-trees in the dependency trees that indicate patterns relating the training pairs. The sub-trees are then used to derive generic indicative patterns (that are independent of the specific training pair instances that discovered the pattern). We then use an appropriate subset of all such patterns discovered to define a set of features that indicate, given a pair of words, the frequency of occurrences in the text corpus of the set of words with respect to each specific pattern. Hence, for any pair of words in the text corpus, we derive a feature vector that is representative of the frequency of occurrences of the pair of words with respect to the patterns. Note that we generate feature vectors for both positive and negative training pairs. This is the most CPU intensive package, as there are thousands of trees that must be analyzed. Therefore, part of this package is threaded so as to make use of a user-defined number of CPUs.

2.5 Package `classifier`

After deriving the feature vector for each training pair, we use the feature vectors to train the classifier. Using word pairs from the corpus and their corresponding feature vectors, we then use the classifier to "induce" new pairs. The classifiers already implemented in the package are logistic regression, Naïve Bayes, Support Vector Machines[1], and a "Naïve Entropy Score"[2] classifier that we designed.

This, again, is another flexible part of the system in that the user can "plug-in" any classifier he/she would like to use.

2.6 Feedback Loop

The induced pairs generated by the classification stage are added to the set of training pairs and the process is repeated (Figure 3). This loop can be run until degradation in quality of relation induction is observed.

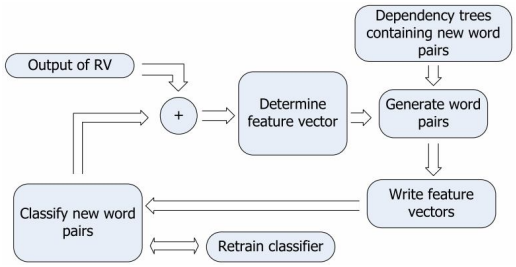

Figure 3: Feedback loop showing how newly found word relationships are used to retrain the classifier.

2.7 Additional Packages

In addition to these four principal packages, we have also included a skeleton package GUI that may implement a set of front-ends that would allow exploiting GARI's functionality in a user-friendly manner.

3 Methodology and Results

3.1 Deriving Training Data

We derived positive training pairs by generating all of the word pairs in WordNet with the given relation and then removing the pairs that do not occur in the corpus. We derived negative training pairs by generating all possible noun pairs from the corpus[3] and then testing them on WordNet to filter out pairs that were related by the selected relationship.

3.2 Deriving Patterns

We used two types of patterns in our testing: patterns based on conventional dependency trees and patterns based on context-free phrase structure trees (also known as Penn trees). In both cases we used a tree search, combined with several pattern matching steps to derive the pattern. We also included several preprocessing stages including word stemming. The Penn tree method seemed to produce more consistent patterns than the dependency tree method and also required less preprocessing. However, the Penn trees required significantly more physical memory during the computations and produced longer pattern strings.

[1] Using LibSVM 2.85

[2] This classifier works as follows: for each relation pair, each pattern in the feature vector is assigned a score in the range [-1,1], depending on how indicative/anti-indicative the pattern is of the given relationship (where zero indicates little information about the pattern for that relation pair). Classification is done by summing over the patterns' scores weighted by the pattern's frequency of occurrence.

[3] We used a corpus consisting of nuclear development abstracts from 1986 through 2001 from the Center for Nonproliferation Studies (CNS) at the Monterey Institute of International Studies. The corpus was obtained through the AQUAINT Program and contains around 300,000 sentences.

We also found that our most frequent patterns were "well known" patterns. For example, for the hypernym relation, the two most common patterns were "WORD2 and other WORD1" and "WORD1 such as WORD2", which agree with the commonly used Hearst [3] patterns for hypernym relation induction.

3.3 Deriving Useful Patterns

We used the n most commonly occurring patterns as our "useful" patterns. Restricting the number of patterns (i.e., the number of features) mitigated the effects of overfitting, but also resulted in some word pairs not being discovered later on. To find the optimal value of n, we ran our classifiers for different values of n (where n was swept between 5 and 500) and recorded the accuracy and recall from each run.

Varying n had the most effect on recall. Recall generally decreases as n increases. This is most likely due to the fact that as n increases, our classifier model becomes more complex and starts to overfit the training data. In turn, more relation pairs in the test set fall into the "unknown" category, decreasing the recall. Recall is a function of the size of the test set minus the number of relation pairs classified as "unknown."

However, since using too low of values for n would lead to low accuracy, a tradeoff must be made between these two factors. In our tests, for training on 100 Penn trees, we found n = 50 to yield reasonable accuracy without a significant drop in recall.

3.4 Generating Features

We used the number of times each unknown relation pair matched each pattern as the feature vector (with one feature for each pattern).

3.5 Testing

We successfully ran our algorithm for training set sizes up to 5000 Penn trees. However, we were able to compile significant statistics for testing done only with 50 to 100 Penn trees (about 500-1000 word pairs) in the training set. Full testing on larger training sets resulted in infeasibly large computation time.

Classifier	Positive Relation Accuracy	Negative Relation Accuracy	Recall (POS:NEG)
Naïve Bayes w/ margin	95.5%	99.0%	0.007 : 6.163
Naïve Entropy w/ margin	50%	94.1%	0.007 : 0.005
SVM	32.4%	100%	0.99 : 6.447
Log-Reg w/ margin	23.3%	97.5%	1.279 : 6.695

Table 1: Accuracy and recall results for each classifier on a single 50-sentence corpus. Recall calculated for each labeling as the number of induced pairs not in the training set divided by the training set size.

We learned from our experiments that using the Penn trees resulted in much better test accuracy than with the dependency trees. We tested our system by hand tagging about 5000 randomly chosen word pairs from the corpus for the hypernym relation. We then compared these hand-tagged pairs against the induced relation pairs[4].

Although our training sets and test sets were limited in size, we can still infer information about the classifiers relative to each other. Since all classifiers had more than an order of magnitude ratio in negative-to-positive training pairs, all classifiers induced a large number of negative relations that were virtually all correct. The accuracies in Table 1 are for correct labeling of relation pairs classified as "yes" by each classifier, as compared to our hand-labeled list.

Naïve Bayes had the best accuracy of all the classifiers. It is well suited to this sort of application, where we have very sparse feature vectors consisting of small positive numbers. Naive Bayes had to be modified for this use by putting a margin between the "yes"/"no" labelings, leaving a group of "unknown" labelings in the middle.

Generally, accuracy tends to decrease as recall increases. In addition to characteristics of the individual classifiers, recall and accuracy tend to have an inverse relationship because the accuracy is partially set by the classifier's margin. As the margin is decreased, the recall will increase, but the accuracy will be lower because pairs that had

[4] Accuracy was calculated only from the subset of induced pairs that were in the hand-tagged set.

been previously classified as "unknown" are now being set as "yes" or "no."

Logistic regression gave the lowest accuracy of all the classifiers. In general, it is not well suited to this type of application due to its Hessian matrix being very sparse[5].

3.6 Inducing New Relation Pairs

For each iteration of the feedback loop, the algorithm was able to induce around 5% of the original training set size (using a larger training set and small feature vector with the Naïve Entropy Score Classifier). In other words, on each iteration, the number of relation pairs classified as "yes" or "no" (i.e., not "unknown") increased the training set for the next iteration by 5%.

Despite these very positive results, the induction step was also very processor intensive and time consuming, and hence prevented us from compiling a larger set of test results. However, parallelizing several computation steps (in package `feature`) did give some improvement in performance. We threaded our algorithm to work across multiple machines, and were able to achieve significant speedup for up to 8 machines (Figure 4).

3.7 Terminating Feedback Loop

Each classifier terminates its feedback loop when the classifier begins to induce incorrect relation pairs. This can be observed by the fact that the classifier will start incorrectly classifying known positive and negative relation pairs. Due to computational complexity, we were not able to run the feedback loop enough iterations to induce a good breaking point. However, this can be set by the user to meet accuracy constraints.

4 Future Work

There are 2 primary objects that we recommend as the next steps for this project: 1) generalizing the relationship extraction algorithm, and 2) optimizing the algorithm for speed. Generalizing the system includes making it capable of inducing across-paragraph and across-multiparagraph relationships. Certain word relationships, such as verb-verb relationships, are difficult to induce from single-sentence parsing because multiple verbs in a sentence do not occur as frequently as multiple nouns.

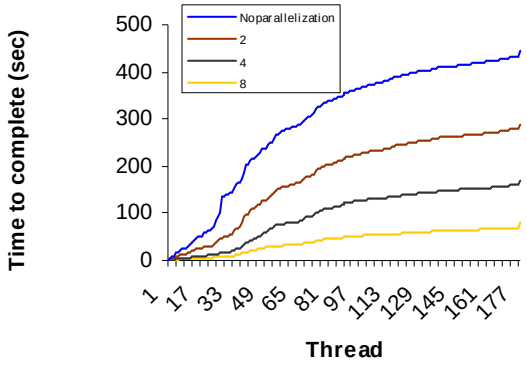

Figure 4: Execution time

Our other recommendation is to work on optimizing the code such that it is not as processor intensive. This specifically applies to the `feature` package, which in the current state would require several CPUs to be feasible in a real-world application. This optimization most likely would include using a different form of storing the decision trees such that extracting patterns and new word pairs from the corpus is less CPU intensive.

5 Acknowledgements

We would like to thank Stanford graduate student Rion Snow, who advised us and shared his work with us during this project. We would also like to thank Profs. Dan Jurafsky and Andrew Ng for advice they gave us along the way.

6 References

[1] Snow, R. & Jurafsky, D. & Ng, A. Y. (2005) *Learning syntactic patterns for automatic hypernym discovery.* NIPS 2005.

[2] de Marneffe, MC. & MacCartney, B. & Manning, CD (2006) *Generating Typed Dependency Parses from Phrase Structure Parses.* Proceedings of 5th International Conference on Language Resources and Evaluation (LREC2006). Genoa, Italy.

[3] Hearst, M. (1992) *Automatic Acquisition of Hyponyms from Large Text Corpora.* Proc. of the Fourteenth International Conference on Computational Linguistics. Nantes, France.

[5] We used a modified version of logistic regression where we set all zero values in the Hessian to very small positive values.