Maximum Entropy Part-of-Speech Tagging in NLTK

Gregory Malecha   Ian Smith

May 14, 2010

Abstract

In this paper we implement a part of speech tagger for NLTK using maximum entropy methods. Our tagger can be used as a drop-in replacement for any of the other NLTK taggers. We give a brief tutorial on how to use our tagger as well as describing the implementation at a high level. We evaluate our tagger on the Penn Tree Bank and compare our results to those of previous work.

1 Background

1.1 Existing Work

NLTK has a number of feature set based classifiers already; these operate on a variety of algorithms, including decision-tree models, Naive Bayesian models, the Mallet and Weka machine-learning package, and maximum entropy models. Some work has already been done to create a part-of-speech tagger in NLTK using maximum entropy models [5] (the most efficient implementation of which seems to be the megam package [2], which NLTK can call out to). Our initial reading of the paper suggested that they had used a manually-selected feature set; as it turned out, this was not true, and our plan (to use the most common suffixes of length $\leq N_1$ for some N_1, and the tags of the preceding N_2 words) was almost identical to their implementation. (We also experimented with adding prefixes, with the aim of seeing whether prefix features would be informative.)

1.2 Maximum Entropy Modeling

A maximum entropy model seeks to generate a classifier model for a set of features. By maximizing the entropy in our model, we are attempting to minimize the amount of information the model carries; information, in this case, in the form of assumptions about our language model that the data do not support. We generate a feature set for our language model (one such feature could be, "does the word end in suffix S?"), find the values of the feature set for our training data, optimize the weights of our features to maximize the entropy, and then we have a language model. We can feed our language model a set of features associated with a given token we wish to classify, and the system can then give us the probability that our token falls into any given class of tokens against which our language model was trained. For a more detailed description of maximum entropy ('maxent') modeling, see [1].

2 Implementing the Maximum Entropy Tagger

The NLTK [3] system defines a generic part-of-speech tagging interface as well as a library for training maximum entropy classifiers based on input data. In this section we cover our implementation interface that connects these two components and give a simple example for how to use it.

2.1 Interface

The part-of-speech tagger interface is defined in nltk.tag.TaggerI [1]. In order to provide a plug-in replacement for the nltk framework, we define an instance of this interface MaxEntTagger which implements the tag method. This tagger implementation takes a language model which is trained from a candidate feature set and tagged training corpus.

Feature Sets

The FeatureSet interface encapsulates a set of named features which are defined by functions from sequences of words to boolean values. For example, a features that checks whether the target word ends in `-ed` or `-ing` could be implemented as: [2]

```
('endswith-ed', lambda w : word(w).endswith('ed'))
('endswith-ing', lambda w : word(w).endswith('ing'))
```

The FeatureSet abstraction takes a set of tokens (tagged or untagged) and converts each token into a dictionary from feature names to feature values where the feature value is the result of applying the second component of the feature tuple above to the token.

Language Models

LanguageModels are built from FeatureSets by training the feature set on a tagged corpus and building a classifier by invoking NLTK's MaxentClassifier.train function. This can take a considerable amount of time, so it is recommended that LanguageModels be serialized to disk (using the Python pickle module) and then reloaded when they are needed.

The Tagger

Given a LanguageModel, the MaxEntTagger uses it to infer tags for untagged sentences. The class implements the TaggerI interface and can therefore be used as a drop-in replacement for any of the other taggers that NLTK provides, e.g. the Brill tagger or one of the n-gram taggers.

2.1.1 Skeleton Tagger Usage

Figure 1 provides the skeleton of a script that uses our MaxEntTagger. The first four lines partition the corpus into a training set and a testing set. In the example we use the Penn Tree Bank, but any of the tagged corpora should work. Lines 6 through 12 construct a feature set based on the prefixes and suffixes that occur in the training data. prefixFeatures constructs a FeatureSet with features for a word starting with each string in the list; the affixes function extracts the list of prefixes up to length 2 from the given corpus. suffixFeatures does the same for word suffixes. The JoinFeatureSet just constructs a FeatureSet that is the union of the given FeatureSets.

Line 15 trains a language model using the training corpus. Line 18 is optional but convenient; we use Python's pickle module to serialize the learned language model since training a language model even from relatively few features on relatively few examples can take a considerable amount of time. We construct the tagger in line 21 and use it to tag the test corpus on line 24.

2.2 Features

Based on previous work, we consider several types of features, which are likely suitable for different languages.

[1] `nltk.googlecode.com/svn/trunk/doc/api/nltk.tag.api.TaggerI-class.html`

[2] Due to restrictions in Python on serializing lambdas, it is recommended that closures used for this purpose are rewritten with classes that override the `__call__` method.

```
# Get the corpus
corpus = treebank.tagged_sents()
training_corpus = corpus[:500]
test_corpus = corpus[501:1500]

# Construct the FeatureSet
feature_set = JoinFeatureSet(prefixFeatures(affixes(training_corpus,
                                                    length=2,
                                                    prefix=True)),
                             suffixFeatures(affixes(training_corpus,
                                                    length=2,
                                                    prefix=False)))

# Train the LanguageModel
lang_model = LanguageModel(feature_set, training_corpus)

# It is usually a good idea to pickle this, since training can take a while
pickle.dump(lang_model, open('penn500.lm', 'w'))

# Construct a Tagger from the language model
tagger = MaxEntTagger(lang_model)

# Run the tagger on the untagged sentences
tagger.batch_tag([nltk.tag.untag(sent) for sent in test_corpus])
```

Figure 1: Skeleton script for using the `MaxEntTagger`.

2.2.1 Affix Features

The simplest type of feature that we use are affix features. These features are based on the prefixes and suffixes of a word. These features are likely to be most useful in languages that use morphological rules to modify words. For example, many European languages are based on declensions of nouns and adjectives and conjugations of verbs. In these cases, we expect common declension or conjugation endings to become noteworthy features suggesting nouns or verbs respectively.

We construct these features automatically from the training corpus by recording all prefixes and suffixes up to a certain length.

2.2.2 Neighborhood Features

In addition to using the current word, we can also use the tags of surrounding words as features. A common example in English might be that the word following a determiner is often a noun and sometimes an adjective, but rarely a verb or preposition. We expect these features to be useful to classification when languages use modifiers and word positioning to convey meaning.

2.3 Implementation

The basic implementation of the tagger simply applies the feature set to the sentence the same manner that was used for training, and passes the resulting features off to the trained language model. This works fine when all of the features are only based on the current word, but this is not the case when neighborhood features are used. There are several strategies for determining the tag for a word when the tags of surrounding words contribute to the features of a word.

	Three	computers	that	changed	the	face	of	personal	computing
1	CD	NNS	IN	VBN	DT	NN	NN	JJ	VBG
2	CD	NNS	IN	VBN	DT	NN	IN	NN	NN
3	CD	NNS	IN	VBD	DT	NN	NN	JJ	VBG
4	CD	NNS	IN	VBN	DT	NN	IN	NN	NN
5	CD	NNS	IN	VBD	DT	NN	NN	JJ	VBG

Figure 2: An example of a sentence fragment whose labels do not converge in the iterative algorithm.

- **Incremental** In this case we begin by using only features which are based on the current word. After we have bootstrapped the process with these tags, we then refine the classifications using the tags in generation i to compute the tags for generation $i+1$. We can iterate this for a fixed number of steps or until a fixed-point is reached. Our experiments suggest that naively doing this does not always result in a fixed-point.
- **Forward** When the neighborhood only mentions the tags of previous words, we can iteratively compute the tags starting from the beginning of the sentence and moving to the right.
- **Backward** When the neighborhood features only mentions the tags of subsequent words, we can iteratively compute the tags starting at the end of the sentence and moving left.

Because we need to handle both previous and subsequent words, we use the incremental strategy to compute the tags.

Non-Convergence of the Iterative Algorithm

Our experiences show that in many cases, the iterative algorithm begins to loop rather than converge to a single value. For example, consider the example sentence in Figure 2. Here, the final three words enter a loop in which using the initial assignment `NN JJ VBG` results in a new assignment `IN NN NN` and vice-versa. Another loop occurs in the classification of the word "changed". There are several approaches to dealing with this problem, however we did not have time to test any of them. We believe that the most principled approach would be to return a probability distribution of tags for each word rather than returning a single tag.

3 Testing the Tagger

We tested the tagger using cross validation on the Penn Treebank [4]. The biggest difficulty in evaluating the tagger was the time and memory that training took. We broke the tree bank into 10 equal-sized chunks and trained on 4 parts and used the remainder for testing. Ideally we would have used used 9 parts for training, but the maximum entropy optimization problem requires considerable memory resources (almost 4GB for training a model on 4 parts). In addition to high memory requirements, the computational burden is also significant. Training a language model on 4 chunks of the tree bank takes several hours even on a 3Ghz machine and, unfortunately is completely serial [3].

The candidate features that we use in our trials are given in Figure 3. The first three features are prefix features and are added for every prefix of length 1, 2 and 3 that occurs in the training corpus. These features evaluate to true if the word begin tagged starts with the given prefix. The next three features are suffix features of lengths 1, 2, and 3. These are analogous to the prefix features. The remaining six features a neighborhood features. They evaluate to true if the tag of the word at a given offset is equal to the tag T.

[3] We trained our models using the megam optimizer since it is written in OCaml which is considerably faster than Python.

Feature Name		Condition	Meaning
`pre-1-`x	=	word[0] == “x”	Word begins with “x”
`pre-2-`xx	=	word[0:1] == “xx”	Word begins with “xx”
`pre-3-`xxx	=	word[0:2] == “xxx”	Word begins with“xxx”
`suf-1-`x	=	word[-1] == “x”	Word ends with “x”
`suf-2-`xx	=	word[-2:] == “xx”	Word ends with “xx”
`suf-3-`xxx	=	word[-3:] == “xxx”	Word ends with “xxx”
`tag-`T`@-1`	=	tag[-1] == T	Previous word has tag T
`tag-`T`@-2`	=	tag[-2] == T	Two words back has tag T
`tag-`T`@-3`	=	tag[-3] == T	Three words back has tag T
`tag-`T`@+1`	=	tag[1] == T	Next word has tag T
`tag-`T`@+2`	=	tag[2] == T	Two words forward has tag T
`tag-`T`@+3`	=	tag[3] == T	Three words forward has tag T

Figure 3: Feature set used for our results.

3.1 Results

Figure 4 shows the average accuracy of our models. Surprisingly some of what we thought would be some of the easiest results to tag came out pretty bad. For example, the tag **TO** (the word “to”) has a very high error rate. This might be fixable by adding a feature such as “is the word *w*.” It would then be up to the training algorithm to determine the significance of this tag. Several other parts of speech were generally tagged poorly including **FW** (foreign word), **LS** (list item separator), **SYM** (symbol) and **UH** (interjection). With the exception of **TO** and possibly **FW** these are relatively rare so it might simply be due to a lack of training data to properly learn these classes.

The remainder of the tags fair reasonably well with most getting an accuracy over 70%. Perhaps somewhat surprisingly, the tagger does fairly well on most of the variations on nouns and verbs with the exception of **NNPS** (plural proper nouns), **VBZ** (3^{rd} person, present, singular verbs), and **VB** (verb base form).

To get an idea of the types of mistakes that the tagger is making, consider the confusion matrix in Figure 5. The confusion matrix shows what types of errors the tagger makes; for example, the number in the cell **VB**,**VBG** is the number of times that the tagger misclassified a **VB** as a **VBG**. Thus, if our tagger was perfect all of the off-diagonal elements would be 0, as the figure shows, this is not the case. As can be seen by the graph, our classification is reasonable for most of the syntactic classes. As we discussed previously there are several tags that are very bad, for example **TO** which gets almost none correct. From the confusion matrix we note that the tagger is concentrating this area on **JJ** (adjectives) and **NN**. This could be due to the fact that the tagger is mistaking indirect objects for direct objects. Other notable deviations occur in **SYM** (symbols) which seem to be mostly classified as **NN** (nouns) as well. Somewhat surprisingly **LS** (list separator) is most commonly classified as a **CD** (cardinal number) and almost never as a symbol [4]. The only reasonable explanation for this is that these tags are mostly affected by the neighborhood features since most words tagged as **CD** are numeric, though they could also be number words.

In addition to looking at the confusion matrix, we also consider the words that the parser misclassifies most often. These results are given in Figure 6. The majority of our mistakes here are on very frequent words. This reinforces our previous speculation that adding the literal word as a feature would greatly improve our accuracy, assuming that the trainer could learn a high enough weight for these literal features. It also suggests that simply adding an extra layer to correct these common words has potential to drastically improve the accuracy of the parser.

Another thing that is interesting is that the commonly misclassified words are spread among many categories; i.e. the tagger is not consistently misclassifying to a small number of incorrect tags. This suggests that these words occur in many different contexts that are affecting the tagging in significant ways.

[4]There is a special class **,** for commas that doesn’t occur in the standard list of Treebank tags but does occur in the taggings exported by NLTK.

Tag	**Occurrences**	**Percent Correct**
CC	15761	88.01%
CD	23917	99.33%
DT	53985	97.49%
EX	553	96.20%
FW	121	11.57%
IN	64859	60.79%
JJ	39945	81.08%
JJR	2262	79.22%
JJS	1274	89.80%
LS	33	9.09%
MD	6284	62.75%
NN	87651	89.19%
NNS	39752	94.38%
NNP	60294	96.18%
NNPS	1660	50.12%
PDT	193	86.53%
POS	5579	92.29%
PRP	11564	43.77%
PRP$	5542	95.38%
RB	20188	80.45%
RBR	1081	61.70%
RBS	287	77.35%
RP	1704	31.63%
SYM	37	10.81%
TO	14748	0.00%
UH	56	10.71%
VB	17266	57.90%
VBD	20125	93.08%
VBG	9616	92.59%
VBN	13335	83.46%
VBP	8219	84.41%
VBZ	14069	57.57%
WDT	2823	96.03%
WP	1584	97.54%
WP$	100	100.00%
WRB	1357	98.75%
Total	547824	81.57%

Figure 4: The average accuracy of the tagger for each tag for one of the models.

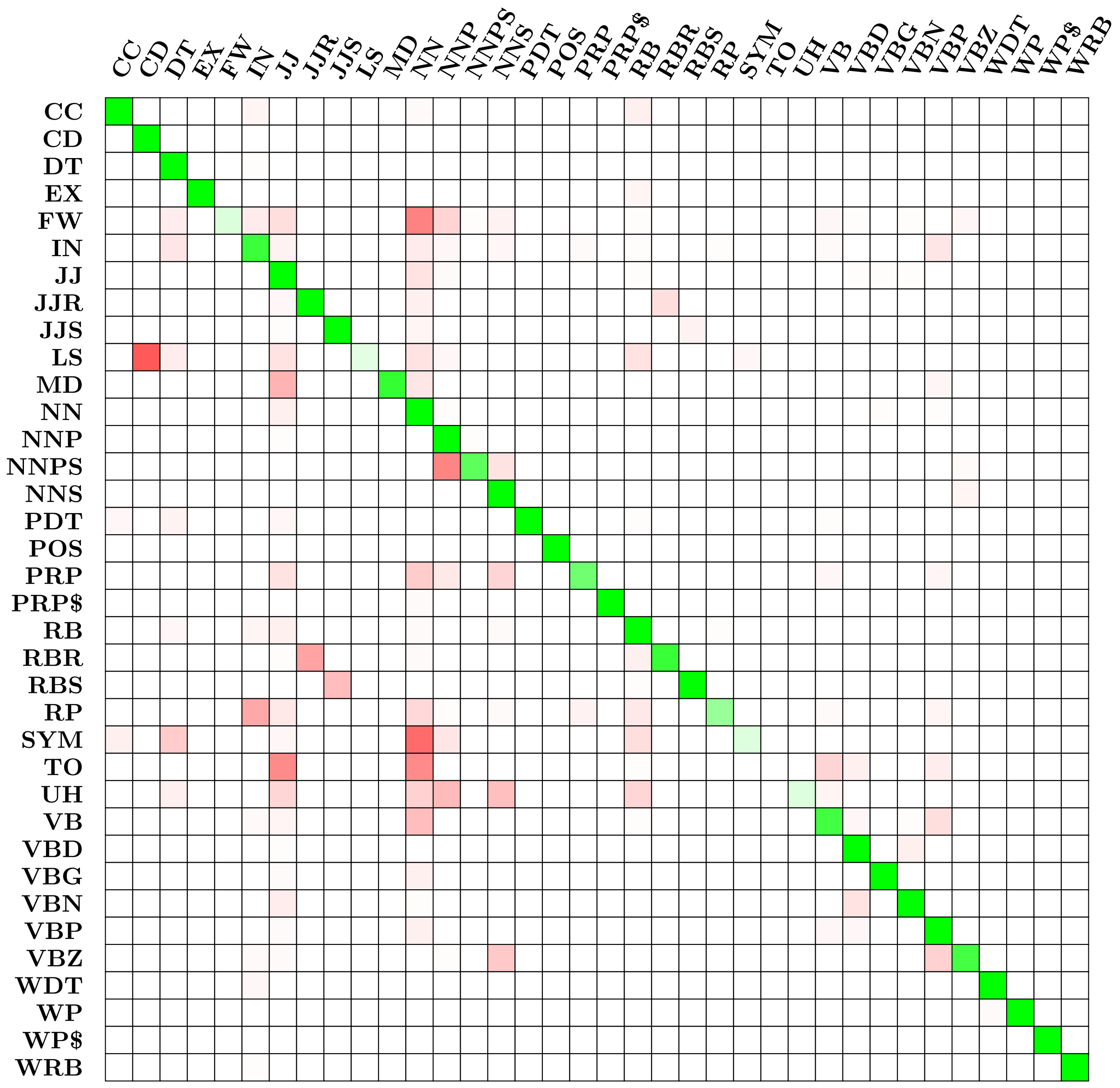

Figure 5: Confusion matrix for one of the results.

	Word	**Occurrences**	**% Error**	**Misclassifications by Frequency**
1.	to	14639	99.99%	JJ, NN, VB, VBP, VBD, RB, NNP, VBN, MD, PDT, FW
2.	he	1785	99.94%	NN, NNP, JJ, RB, VBP, VB, CD, LS
3.	is	4327	99.93%	VBP, NNS, JJ, IN, NNP, VB, NN, PRP, FW, RP, RB, PDT, VBZ, VBN
4.	at	2842	99.93%	DT, VBP, NN, NNP, VB, IN, RP, PDT
5.	In	1153	99.91%	IN, VBP, NNS, JJ, PRP, NNP, VB, RP, FW, NN, PDT, RB, VBN, CD, MD, NNPS, VBG
6.	as	2737	99.85%	DT, NN, VBP, VB, NNP, IN, VBD
7.	He	571	99.82%	NN, NNP, JJ, RB, VBP, VB, CD, LS
8.	we	451	99.78%	NNP, NN, JJ, MD, VB, VBP, PRP, VBD, WRB, PDT
9.	or	1762	99.77%	RB, IN, NN, VBP, JJ, NNP, VB, CD, MD, RP, CC, NNS
10.	so	417	99.76%	NNS, VBZ, NNP, JJ, PDT, IN, VB, RB
11.	If	235	99.57%	PRP, VBP, JJ, IN, NNS, VB, NNP, RP, PDT, NN, FW, RB
12.	As	206	99.51%	DT, NN, VBP, VB, NNP, IN, VBD
13.	my	170	99.41%	NN, NNP, JJ, VBP, MD, VB
14.	by	2913	99.31%	NN, VB, JJ, VBP, NNP, RP, IN, VBD, NNPS, CC, VBN, CD, RB
15.	At	223	99.10%	DT, VBP, NN, NNP, VB, IN, RP, PDT
16.	To	105	99.05%	JJ, NN, VB, VBP, VBD, RB, NNP, VBN, MD, PDT, FW
17.	me	95	97.89%	NN, VBP, JJ, VB, PRP
18.	We	378	97.09%	NNP, NN, JJ, MD, VB, VBP, PRP, VBD, WRB, PDT
19.	My	32	96.88%	NN, NNP, JJ, VBP, MD, VB
20.	No	56	96.43%	JJ, RB, NN, VBP, NNP, VB, IN

Figure 6: Top 20 most difficult words for the tagger to correctly classify.

Feature Name		Condition	Meaning
`pre-1-`x	=	word[0] == “x”	Word begins with “x”
`pre-2-`xx	=	word[0:1] == “xx”	Word begins with “xx”
`pre-3-`xxx	=	word[0:2] == “xxx”	Word begins with“xxx”
`pre-4-`$xxxx$	=	word[0:3] == “xxxx”	Word begins with“xxxx”
`suf-1-`x	=	word[-1] == “x”	Word ends with “x”
`suf-2-`xx	=	word[-2:] == “xx”	Word ends with “xx”
`suf-3-`xxx	=	word[-3:] == “xxx”	Word ends with “xxx”
`suf-3-`$xxxx$	=	word[-4:] == “xxxx”	Word ends with “xxxx”
`number`	=	any([d in word for d in digits])	Word contains a digit
`uppercase`	=	not word.lower() == word	Word contains an uppercase character
`hyphen`	=	’-’ in word	Word contains a hyphen
`tag-`T`@-1`	=	tag[-1] == T	Previous word has tag T
`tag-`T_1T_2`@-1-2`	=	tag[-1] == $T_1 \wedge$ tag[-2] == T_2	Previous two tags are T_1 and T_2
`word-`w`@-1`	=	words[-1] == w	Previous word is w
`word-`w`@-2`	=	words[-2] == w	Two words back is w
`word-`w`@+1`	=	words[+1] == w	Next word is w
`word-`w`@+2`	=	words[+2] == w	Two words forward is w

Figure 7: Features used in Ratnaparkhi’s experiment [5].

Also, not that they are all two letter words. This suggests that perhaps prefixes and suffixes are unable to capture the salient properties of these short words like we had hoped because many words might start with “to”, such as “today” or “tomorrow” or end with “is”, such as “crisis” or “hypothesis.” This suggests even further that we need an entire word feature.

3.2 Comparison to Previous Work

Ratnaparkhi [5] performed a similar experiment to ours and reported 96.4% accuracy on words that had occurred in the training corpus and an 86.2% accuracy on words that were new in the test set. In his experiment, Ratnaparkhi used the features given in Figure 7. In addition to this extended feature set, Ratnaparkhi also used a special case heuristic for common words. If a word occurred more than 20 times in the training corpus, a special literal feature was added for that word.

The addition of the literal tokens explains the rather considerable difference between our own results and those of Ratnaparkhi. We believe that because our features included no literal words, our results are best compared to Ratnaparkhi’s results on unseen words. This number is relatively close to our own results.

In retrospect, Ratnaparkhi’s inclusion of the literal words as features in addition to the tags makes sense and seems to correspond closely to the extension from simple part-of-speech parsers to lexicalized parsers. Also, the inclusion of the conjunction tag rule is probably also essential to making use of tags from words that are farther back than the immediately previous word. Finally, we note that Ratnaparkhi’s feature set is evaluatable with the simple **Forward** algorithm described previously (Section 2.3).

4 Conclusions

We developed an implementation of a part-of-speech tagger in NLTK based on maximum entropy models. Our results are comparable to those of previous work for out of vocabulary words since our models don’t do anything special for entire words that it sees in the corpus.

Future Work

We ran into difficulties training and evaluating on the Penn Tree Bank due to its size. Training took 2-3 hours per model and used upwards of 3.5 GB of RAM even using the optimized version of the megam optimizer. This prevented us from finishing several experiments that we were hoping to run.

- Training using the exact feature set that Ratnaparkhi used.
- Adding literal words to our feature set.
- Adding conjunctive tag features (e.g. previous two tags are T_1 and T_2) to our training set.
- Try to precisely quantify the benefit of using the tags of subsequent words as features.
- Evaluation on non-English languages with different morphological processes and syntactic constructions.

References

[1] Adam L. Berger, Vincent J. Della Pietra, and Stephen A. Della Pietra. A maximum entropy approach to natural language processing. *Comput. Linguist.*, 22(1):39–71, 1996.

[2] Hal Daumé III. Notes on CG and LM-BFGS optimization of logistic regression. Paper available at `http://pub.hal3.name#daume04cg-bfgs`, implementation available at `http://hal3.name/megam/`, August 2004.

[3] Edward Loper and Steven Bird. Nltk: The natural language toolkit. *CoRR*, cs.CL/0205028, 2002.

[4] Mitchell P. Marcus, Beatrice Santorini, and Mary A. Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1994.

[5] Adwait Ratnaparkhi. A maximum entropy model for part-of-speech tagging. 1996.