Discriminative Part-of-Speech Tagging

Maxwell Pickering

December 12, 2023

1 Introduction

In this project, I revisit part-of-speech tagging. Previously, we took a generative approach to part-of-speech tagging. We made the Markov assumption that the probability of the emission of a word is solely determined by its part-of-speech tag and the part-of-speech tag of the prior word. Then, using the Viterbi algorithm, we determined the most probable part-of-speech sequence for a sentence, given only the observed words.

It is easy to identify available information beyond what our Markov assumption considers that could be useful in part-of-speech tagging, such as word suffixes and words appearing later in the sequence. However, integrating this information into a generative tagging model requires coming up with a probabilistic model of the emission of the word based on all these facts. This is difficult.

A discriminative approach allows us to incorporate additional features without inventing such a generative story. In this project, I used conditional random fields to perform part-of-speech tagging on English sentences. I took advantage of the flexibility of this discriminative method to experiment with feature representation, and achieved an accuracy of 0.93, relative to an accuracy of 0.86 from always guessing the most common tag for a given word.

2 Data

The POS-tagged English sentences necessary for this project were obtained from Universal Dependencies' English Web Treebank dataset. This data consists of 16,622 sentences from blogs, news, emails, reviews, and Yahoo! answers, collected around 2012. The data is pre-partitioned into training, development, and testing sets.

Partition	Sentences	Tokens
Training	12,544 (75.5%)	204,607 (80.3%)
Development	2,001 (12.0%)	25,151 (9.9%)
Testing	2,077 (12.5%)	25,096 (9.8%)
Total	**16,622**	**254,854**

Table 1: Dataset Distribution

I retrieved the data in the .conllu format, and used the package *pyconll* to extract the token and tag sequences. I skipped over untagged tokens: these were all instances where a contraction was tokenized as two tokens, but given only one POS tag. Then, I was left with nice, tagged English sentences such as these:

```
Chalabi has been increasingly marginalized within Iraq  ,     however ,
PROPN   AUX AUX  ADV          VERB         ADP    PROPN PUNCT ADV     PUNCT
despite his  ties of  clientelage with Washington and   Tehran .
ADP     PRON NOUN ADP NOUN        ADP  PROPN      CCONJ PROPN  PUNCT

He   is  no  longer in  the dominant Shiite list ,     the United Iraqi Alliance
PRON AUX ADV ADV    ADP DET ADJ      ADJ    NOUN PUNCT DET ADJ    ADJ   PROPN
,     and   wo  n't  have many seats in  the new parliament .
PUNCT CCONJ AUX PART VERB ADJ  NOUN  ADP DET ADJ NOUN       PUNCT
```

The data was manually annotated in the Penn treebank POS scheme, and then automatically converted to Universal Dependencies' POS scheme, with manual corrections.

Tags	Description	Count
ADJ	adjective	16,808 (6.60%)
ADP	adposition	21,860 (8.58%)
ADV	adverb	12,521 (4.91%)
AUX	auxiliary	15,938 (6.25%)
CCONJ	coordinating conjunction	8,204 (3.22%)
DET	determiner	20,097 (7.89%)
INTJ	interjection	928 (0.36%)
NOUN	noun	43,165 (16.94%)
NUM	numeral	5,051 (1.98%)
PART	particle	7,044 (2.76%)
PRON	pronoun	23,025 (9.03%)
PROPN	proper noun	16,091 (6.31%)
PUNCT	punctuation	29,768 (11.68%)
SCONJ	subordinating conjunction	4,621 (1.81%)
SYM	symbol	913 (0.36%)
VERB	verb	27,909 (10.95%)
X	other	911 (0.36%)

Table 2: Part-of-Speech Tags Distribution

As with all POS tagging tasks, we note that there is a serious imbalance in the occurrence of different classes. Another important metric to examine specifically for POS tagging is the diversity in POS usage of the tokens. Here are the results of a baseline classifier on the development set, which always tags based on the most common tag for the given type in the training set:

Tag	**Precision**	**Recall**	**F1-Score**	**Support**
ADJ	0.90	0.82	0.86	1869
ADP	0.86	0.89	0.87	2038
ADV	0.92	0.74	0.82	1230
AUX	0.92	0.89	0.90	1568
CCONJ	0.99	0.99	0.99	779
DET	0.96	0.97	0.97	1901
INTJ	0.97	0.58	0.73	115
NOUN	0.68	0.93	0.79	4215
NUM	0.94	0.71	0.81	383
PART	0.69	1.00	0.81	647
PRON	0.97	0.94	0.95	2221
PROPN	0.91	0.52	0.66	1786
PUNCT	0.99	0.99	0.99	3075
SCONJ	0.66	0.63	0.64	398
SYM	0.86	0.75	0.80	83
VERB	0.87	0.79	0.83	2709
X	0.46	0.04	0.08	134
Accuracy				0.86 (25151)

Table 3: Most Common Tag by Type Classification

This most common tag by type accuracy of 86% is worse than the 92% accuracy achieved by the same method on the Wall Street Journal corpus, indicating that tokens in this dataset more often occur in parts-of-speech other than their most common category. (Jurafsky) This makes the task more difficult.

3 Feature Development

Of course, there is a giant search space to explore when developing a model, considering all the combinations of feature configurations and hyperparameter settings. To make this manageable, I structured my search by first experimenting with feature configurations on a baseline averaged perceptron model, and then tuning hyperparameters on different models with that fixed feature configuration.

As a baseline, I simply used a lowercased token feature, and a boolean indicator as to whether the token was originally uppercased, so as not to lose information. On the baseline averaged perceptron model, this feature configuration achieves an accuracy of 65.35, which is extremely poor. It is surprising that it is so bad, since we have seen above that there is enough information to get an accuracy of 86% just from tag by token counts, and the averaged perceptron makes multiple passes over all the training data.

Adding in token suffix features improves accuracy greatly. I tested a suite of feature configurations, each encoding different lengths of suffixes. Even adding a single feature for the last letter of the word increases accuracy to 75.59, while adding last letter and last two letter features increases accuracy to 79.71.

Strangely, this pattern of increasing accuracy continues far beyond what we would expect. The feature configuration with 20 added features, representing the 1-length suffix, 2-length suffix, ..., all the way to 20-length suffix, outperforms all configurations before it, achieving 87.23 accuracy. This is especially weird, because most words are shorter than 20 characters, so many of these new features will be completely redundant. I spoke with Prof. Lignos, who suggested that this was likely because redundant features split the magnitude of their weight between them, and since regularization is based on products, these redundant features allow feature weight magnitudes to grow. The consequence of this would be that, while a 20-length suffix configuration is optimal on this averaged perceptron model, across the full algorithm/hyperparameter space, a feature representation with less redundancy would win out.

To compare, I tested feature configurations which captured windows of context around the token, without any suffix information. I ran configurations capturing just the tokens immediately before, just the tokens immediately after, and different windows around the token in both direction. The accuracy of just before and just after feature configurations were 72.29 and 72.74 respectively, down from accuracies of 76.22, 76.01, and 76.55 for windows of width 1, 2, and 3 respectively. It is interesting that either suffix or neighborhood information is enough to get model accuracy up to the high 70s.

In some of the documentation, I saw a bias feature was used. Experimenting with this bias feature only hurt performance, so I elected not to include it. However, I did find that including a boolean 'is digit?' feature improved tagging for the NUM tag, so I included it.

Combining suffix information and neighborhood information gets the highest accuracy. Taking both suffix and neighborhood information up to three, we achieve an accuracy of 85.05. This is the configuration we will fix going forward, as it balances capturing enough information with avoiding redundancy.

4 Model Tuning

Having fixed a feature configuration, we will now focus on tuning a model to maximize development set accuracy. The two algorithms we will explore beyond averaged perceptron are limited-memory BFGS (LBFGS) and Stochastic Gradient Descent with L2 Regularization (L2SGD). For LBFGS, we'll tune the L1 and L2 regularization coefficients, and for L2SGD, we'll just tune the L2 regularization coefficient. To allow for searching a greater number of configurations, I set the max iterations parameters to 100, to limit how long each configuration could run. Here is a table of accuracies for each combination of examined coefficients.

L1 Coefficient \L2 Coefficient	**1.0**	**0.1**	**0.01**	**0.001**	**0.0001**	**0.00001**	**0.0**
10.0	88.70	89.34	89.36	89.40	89.37	89.37	89.36
1.0	91.88	92.61	92.75	92.72	92.66	92.80	92.71
0.1	91.78	92.57	92.93	93.04	93.04	92.98	**93.09**
0.01	91.81	92.45	92.61	92.87	92.96	92.97	92.99
0.001	91.77	92.40	92.46	92.64	92.71	92.96	92.77
0.0	91.79	92.55	92.28	92.19	92.12	91.94	91.90

Table 4: LBFGS Accuracy Values for Different L1 and L2 Coefficients

L2 Coefficient	**Accuracy**
1.0	64.20
0.1	64.20
0.01	64.20
0.001	64.20
0.0001	64.20
1e-05	64.20
0.0	07.10

Table 5: L2SGD Accuracy Values for Different L2 Coefficients

The results for LBFGS are much nicer than those for L2SGD. For LBFGS, the model performs well: better than the baseline with all hyperparameter configurations. The maximum development set accuracy attained is 93.09, with an L1 coefficient of 0.1 and an L2 coefficient of 0.0. The data for L2SGD is much stranger: except for no L2 regularization at all, which performs horribly at 7.10, the L2 coefficient has no effect on accuracy. All L2 coefficients achieve 64.20 accuracy.

I can think of two possible reasons for this. First, the feature representation which was optimal on the averaged perceptron may be poorly designed for L2SGD. Second, perhaps limiting L2SGD training to 100 epochs is substantially damaging to the model's ability to converge. With more time, I'd like to pursue L2SGD further, but based on these results, it seems best to pursue an LBFGS model.

5 Evaluation

Having fixed a feature set, and performed a hyperparameter search, it is time to evaluate our model on the test set. I'll run three configurations on the test set, one for each algorithm. We expect LBFGS with L1: 0.1 and L2: 0.0 to do the best. We'll compare to L2SGD with L2: 1.0, and to averaged perceptron. Here are the results:

	'lbfgs', {'c1': 0.1, 'c2': 0.0}				'l2sgd', {'c2': 1.0}				'ap', {}			
Tag	Precision	Recall	F1-Score	Support	Precision	Recall	F1-Score	Support	Precision	Recall	F1-Score	Support
ADJ	0.90	0.89	0.90	1787	0.40	0.33	0.36	1787	0.75	0.70	0.72	1787
ADP	0.94	0.96	0.95	2033	0.61	0.79	0.69	2033	0.90	0.91	0.90	2033
ADV	0.92	0.91	0.91	1178	0.88	0.15	0.25	1178	0.72	0.86	0.79	1178
AUX	0.98	0.98	0.98	1543	0.63	0.72	0.67	1543	0.93	0.97	0.95	1543
CCONJ	0.99	0.99	0.99	737	0.93	0.75	0.83	737	0.93	0.98	0.96	737
DET	0.98	0.99	0.99	1897	0.58	0.89	0.70	1897	0.97	0.96	0.96	1897
INTJ	0.93	0.80	0.86	120	0.00	0.00	0.00	120	0.48	0.66	0.56	120
NOUN	0.86	0.91	0.89	4137	0.49	0.73	0.59	4137	0.78	0.78	0.78	4137
NUM	0.93	0.98	0.95	542	0.00	0.00	0.00	542	0.67	0.93	0.78	542
PART	0.96	0.99	0.97	649	0.75	0.68	0.72	649	0.84	0.98	0.90	649
PRON	0.99	0.99	0.99	2162	0.63	0.72	0.67	2162	0.90	0.96	0.93	2162
PROPN	0.84	0.74	0.79	1980	0.77	0.01	0.02	1980	0.78	0.34	0.47	1980
PUNCT	1.00	0.99	0.99	3096	0.85	0.90	0.88	3096	0.98	0.99	0.99	3096
SCONJ	0.88	0.81	0.85	384	1.00	0.06	0.11	384	0.76	0.74	0.75	384
SYM	0.80	0.92	0.85	109	0.00	0.00	0.00	109	0.59	0.72	0.65	109
VERB	0.93	0.95	0.94	2606	0.59	0.69	0.64	2606	0.79	0.91	0.85	2606
X	0.91	0.61	0.73	136	0.00	0.00	0.00	136	0.42	0.50	0.46	136
Overall Accuracy: 0.93					Overall Accuracy: 0.61				Overall Accuracy: 0.84			

Table 6: Evaluation Results for Different Algorithms and Configurations

Indeed, we see what we expected. The test set results are very similar to the development set results, indicating that we did not overfit the data, and that our model will generalize to new data. The winner is the LBFGS configuration, which achieved 93% accuracy.

6 Discussion

The results were largely as expected: a broad, but not redundant, feature pool achieved the highest accuracy. It was also not surprising that the LBFGS model performed the best, as this was suggested to me by Prof. Lignos. However, I do believe there is something wrong with my L2SGD hyperparameter search, as it is strange that changing the L2 regularization coefficient had no effect on accuracy.

As discussed above, I had the surprising result that redundant features could increase accuracy on the averaged perceptron, but it turned out that those redundant features hurt accuracy across the full model and hyperparameter space. This indicates the issues with developing a feature set first, and then performing tuning, instead of using an iterative process to refine the model. With more time, there are many more dimensions I could investigate.

The low baseline accuracies of the CRF models surprised me, because as we discussed above, it should be possible to achieve 86% accuracy just from guessing the most probable tag for a type. I don't know if this is normal for CRF models, or if there was an issue in how I structured my feature data. This being said, simply by throwing together some obvious features, we were able to achieve fairly high POS tagging accuracy, with much less work than with a generative approach.

7 Conclusion

We were able to achieve 93% accuracy on the test set using a diverse feature set containing both suffix and neighborhood information on a tuned LBFGS model. We found that a broad, but not redundant, feature set does well for discriminative POS tagging, and that LBFGS far outperforms other models, including L2SGD and averaged perceptron.

With more time, I'd like to iterate more on feature set development using the tuned LBFGS model. I'd also like to experiment with more specific features, like the presence of known derivational and inflectional morphology on the token.

References

[1] Universal Dependencies. *English Web Treebank*. Retrieved from https://universaldependencies.org/.

[2] Jurafsky, D., & Martin, J.H. (2023). *Speech and Language Processing, 3rd Edition (draft)*, Chapter 8. Retrieved from https://web.stanford.edu/ jurafsky/slp3/.

[3] PyConLL. *PyConLL: A Python Library for Parsing CoNLL*. Retrieved from https://github.com/pyconll/pyconll.

[4] Python-CRFSuite. *python-crfsuite: Python binding for CRFSuite*. Retrieved from https://github.com/scrapinghub/python-crfsuite.