Named Entity Recognition - Report on Approaches Used

Authors: Jay Seabrum, Weitung Liao, Cutter Dalton

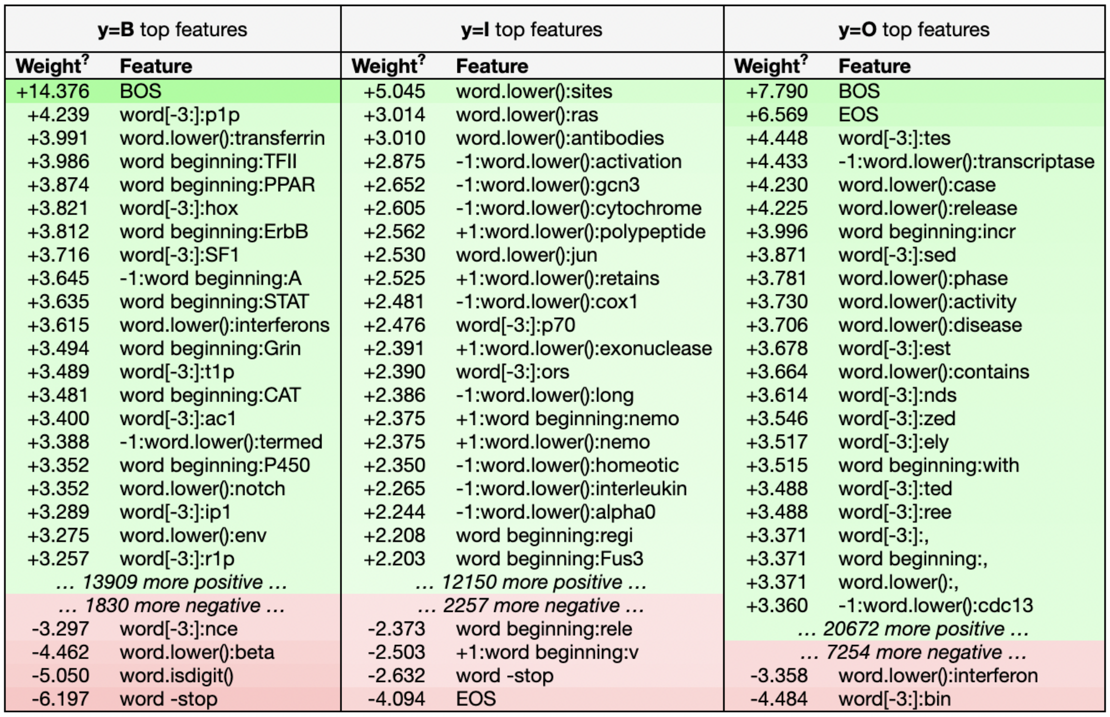
Summary:
Four approaches were taken in this assignment: CRF, Vanilla Features with SGD, Hidden Markov Model, and the “Dumb-As-Bricks” approach. Of these approaches, the best was the CRF approach, yielding a 0.7298 Precision score, 0.5853 Recall score, and a 0.6496 F1 score. This approach’s code is directly submitted with this assignment. The baseline score from the “Dumb-As-Bricks” approach yielded a 0.2625 Precision score, 0.0405 Recall score, and a 0.0701 F1 score. The other two approaches were somewhere between these two, and their code sources are provided within this document.

CRF Approach (Cutter Dalton):

For this section, the code for this approach is submitted alongside this word document.

In this approach, we adopted the crfsuite tutorial for NER to fit this assignment regarding finding genes referenced in biomedical journal articles. We ended up keeping 6 of the features included in the reference code: sentence position, word.lower(), word[-3:], word.isupper(), word.istitle(), and word.isdigit(). This was due to the fact that we believed they each supplied pertinent information for each token. We also added the length of the token, the first four letters of the token, whether it was a stop word, and whether the token had any of the following endings: ‘-ase’, ‘-ene’, ‘-ic’, ‘ein’, and ‘tor’. The specific word endings were added due to a high prevalence of those endings in both B and I labeled tokens.

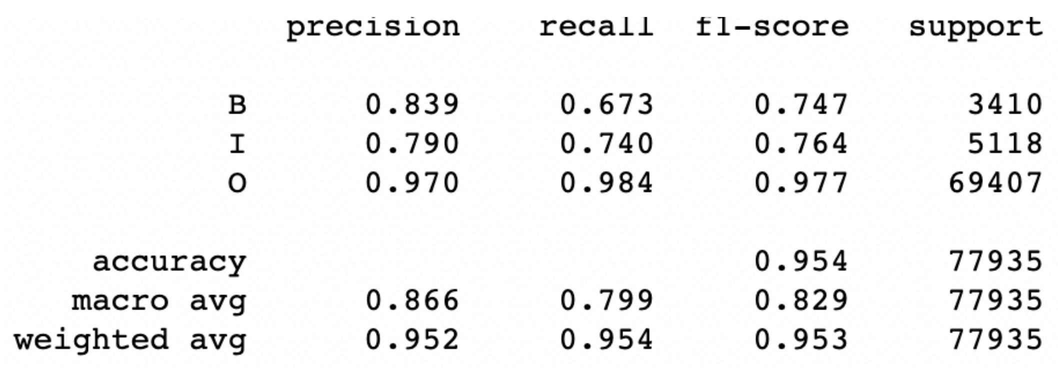
Top 25 weights of the model:



y=**B** top features		y=**I** top features		y=**O** top features	
Weight?	**Feature**	**Weight?**	**Feature**	**Weight?**	**Feature**
+14.376	BOS	+5.045	word.lower():sites	+7.790	BOS
+4.239	word[-3:]:p1p	+3.014	word.lower():ras	+6.569	EOS
+3.991	word.lower():transferrin	+3.010	word.lower():antibodies	+4.448	word[-3:]:tes
+3.986	word beginning:TFII	+2.875	-1:word.lower():activation	+4.433	-1:word.lower():transcriptase
+3.874	word beginning:PPAR	+2.652	-1:word.lower():gcn3	+4.230	word.lower():case
+3.821	word[-3:]:hox	+2.605	-1:word.lower():cytochrome	+4.225	word.lower():release
+3.812	word beginning:ErbB	+2.562	+1:word.lower():polypeptide	+3.996	word beginning:incr
+3.716	word[-3:]:SF1	+2.530	word.lower():jun	+3.871	word[-3:]:sed
+3.645	-1:word beginning:A	+2.525	+1:word.lower():retains	+3.781	word.lower():phase
+3.635	word beginning:STAT	+2.481	-1:word.lower():cox1	+3.730	word.lower():activity
+3.615	word.lower():interferons	+2.476	word[-3:]:p70	+3.706	word.lower():disease
+3.494	word beginning:Grin	+2.391	+1:word.lower():exonuclease	+3.678	word[-3:]:est
+3.489	word[-3:]:t1p	+2.390	word[-3:]:ors	+3.664	word.lower():contains
+3.481	word beginning:CAT	+2.386	-1:word.lower():long	+3.614	word[-3:]:nds
+3.400	word[-3:]:ac1	+2.375	+1:word beginning:nemo	+3.546	word[-3:]:zed
+3.388	-1:word.lower():termed	+2.375	+1:word.lower():nemo	+3.517	word[-3:]:ely
+3.352	word beginning:P450	+2.350	-1:word.lower():homeotic	+3.515	word beginning:with
+3.352	word.lower():notch	+2.265	-1:word.lower():interleukin	+3.488	word[-3:]:ted
+3.289	word[-3:]:ip1	+2.244	-1:word.lower():alpha0	+3.488	word[-3:]:ree
+3.275	word.lower():env	+2.208	word beginning:regi	+3.371	word[-3:]:,
+3.257	word[-3:]:r1p	+2.203	word beginning:Fus3	+3.371	word beginning:,
… 13909 more positive …		*… 12150 more positive …*		+3.371	word.lower():,
… 1830 more negative …		*… 2257 more negative …*		+3.360	-1:word.lower():cdc13
-3.297	word[-3:]:nce	-2.373	word beginning:rele	*… 20672 more positive …*	
-4.462	word.lower():beta	-2.503	+1:word beginning:v	*… 7254 more negative …*	
-5.050	word.isdigit()	-2.632	word -stop	-3.358	word.lower():interferon
-6.197	word -stop	-4.094	EOS	-4.484	word[-3:]:bin

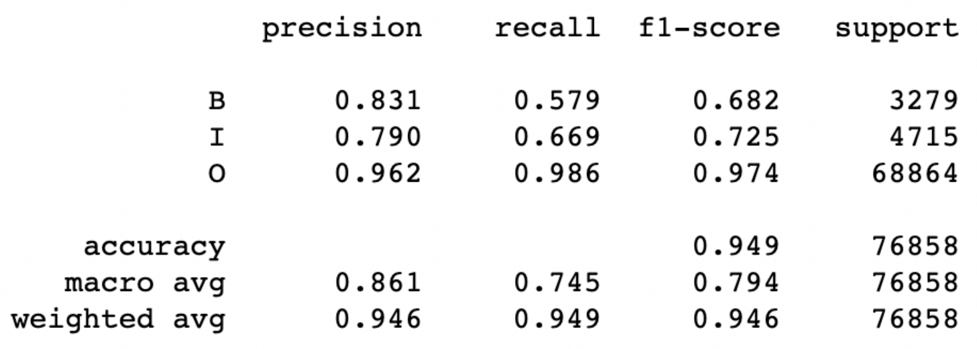
Although we were strongly encouraged to address unknown words, in development testing, we found that not accounting for unknown words actually improved scoring in terms of precision, recall, and F1 for each label. When we did account for unknowns, we assigned words that were not in the training set and did not occur more than 2 times in the development set. Unknown tags were then added to the training set with 5-15% of words assigned the tag at random. Given more time, a better method of accounting for unknowns could be implemented, but given the decent results of the current model, doing so is not incredibly necessary.

Without accounting for unknowns:



```
              precision    recall  f1-score   support

           B      0.839     0.673     0.747      3410
           I      0.790     0.740     0.764      5118
           O      0.970     0.984     0.977     69407

    accuracy                          0.954     77935
   macro avg      0.866     0.799     0.829     77935
weighted avg      0.952     0.954     0.953     77935
```

Accounting for unknowns:

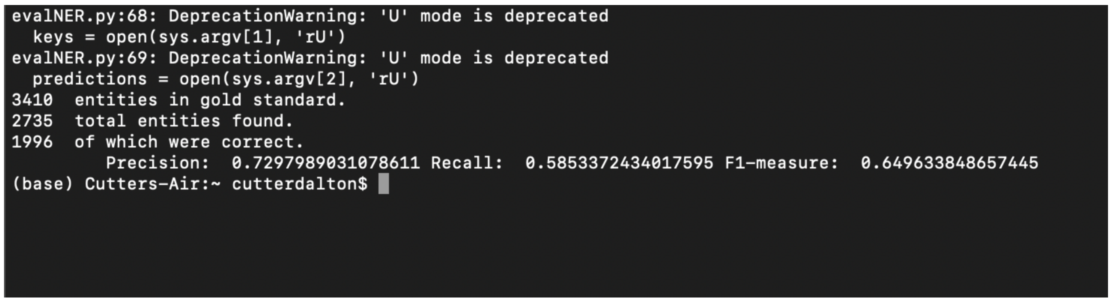


```
              precision    recall  f1-score   support

           B      0.831     0.579     0.682      3279
           I      0.790     0.669     0.725      4715
           O      0.962     0.986     0.974     68864

    accuracy                          0.949     76858
   macro avg      0.861     0.745     0.794     76858
weighted avg      0.946     0.949     0.946     76858
```

In terms of hyper parameters we settled on 200 iterations for the model given that it gave the best results while also completing training in under 2 minutes. We also utilized the standard 80:20 split between training and development data. We also decided to consider the features of the word prior and the word following the word currently being examined in the model.

As seen above, the results were pretty decent for both beginning and inside labels. These results were achieved after tinkering with which features to include and which to discount. Precision for B maxed out at around .840, but its recall suffered, and therefore contributed to a low F1 score at the word level.

Using the evaluation script given by Dr. Martin, our development predictions compared against our development labels yielded the following results:



```
evalNER.py:68: DeprecationWarning: 'U' mode is deprecated
  keys = open(sys.argv[1], 'rU')
evalNER.py:69: DeprecationWarning: 'U' mode is deprecated
  predictions = open(sys.argv[2], 'rU')
3410  entities in gold standard.
2735  total entities found.
1996  of which were correct.
         Precision:  0.7297989031078611 Recall:  0.5853372434017595 F1-measure:  0.649633848657445
(base) Cutters-Air:~ cutterdalton$
```

Referenced Code:
(https://sklearn-crfsuite.readthedocs.io/en/latest/tutorial.html#let-s-use-conll-2002-data-to-build-a-ner-system)

Vanilla Feature-Based Approach with Stochastic Gradient Descent (Jay Seabrum):
For this section, the code for this approach can be found here on Jay's GitHub:
https://github.com/xjseabrum/nlp-files/tree/main/assgn3/feature_extraction.py

To begin, in this approach, the BIO tags were converted into a dummy variable called "bin_tag" which assigned a 1 when the tag was B or I, and 0 when the tag was an O. Additionally, a variable called "group_id" was created that assigned a numerical identification number to words that belonged to the same sentence. This was done to make subsetting the dataframe into training, validation, and holdout subsets easier.

From there, statistical data exploration was performed on all the words in the dataset with respect to bin_tag. While word length proved not to be a valuable feature for these data, some features, however, did look important enough to include:

- "is_camel": This is a dummy variable that is 1 if the word is camel case and 0 otherwise.
- "all_caps": This dummy variable is 1 if the word is all caps or a number and 0 otherwise.
- "ase": This dummy variable is 1 if the word ends in *ase* or *ases* and 0 otherwise.

Next, the words were grouped into 5 bins, from which 4 categorical dummies were made. These 4 categorical dummies represented the following:

- "top_80": 1 if the word is tagged [80 - 100%] of the time, 0 otherwise.
- "top_60": 1 if the word is tagged [60 - 80%) of the time, 0 otherwise.
- "top_40": 1 if the word is tagged [40 - 60%) of the time, 0 otherwise.
- "top_20": 1 if the word is tagged [20 - 40%) of the time, 0 otherwise.

To account for unknown words, words that occurred less than 10 times in the dataset were masked with "<UNK>". Stats on this <UNK> mask revealed that <UNK> words were tagged with a B or an I roughly 21% of the time. This means that in the validation dataframe, words that weren't in the training dataset were assigned a 1 under the top_20 variable.

With these 7 features and the "bias" feature, the data were split by group_id into 72/18/10 training/validation/holdout datasets. The SGD model with a learning rate of 0.1 and 1000 and 5000 iterations per word was trained using the training dataset. The weights learned from the 1000 iteration run and the 5000 iteration run are as follows (rounded to 4 decimal places):

# of Iter. \ Feat.	is_camel	all_caps	top_80	top_60	top_40	top_20	ase	bias
1000	0.2061	2.1137	15.802	7.5434	5.7601	5.6922	-0.745	-9.6803
5000	0.7947	0.9555	28.4555	15.3504	11.3956	8.309	-3.705	-14.7907

One thing to note from these weights is that with more iterations, the weights get quite ridiculous, exceeding ±7 for most of the features. These weights were then used to predict a 1 or a 0 (bin_tag) on the validation dataset. From there, the 1s were labelled as an I and the 0s as an O (called "PredLbl"). Finally, to relabel the tags so that they would be in BIO format, every I that immediately followed an O on the predicted tags was changed to B.

The results from the evalNER.py file for both 1000 and 5000 iterations are as follows (rounded to 4 decimal places):

# of Iter. \ Metric	Precision	Recall	F1
1000	0.4432	0.4242	0.4335
5000	0.4019	0.4732	0.4347

With more iterations, the recall and F1 metrics increased at the expense of precision.

Four other features were tested on these data but were excluded in the final runs of the model due to serious degradation of model performance. These tested features were:

- "ene": 1 if the word ends in *ene*, 0 otherwise.
- "ein": 1 if the word ends in *ein* or *eins*, 0 otherwise.
- "hyp": 1 if the word was just a hyphen, 0 otherwise.
- "tor": 1 if the word ends in *tor* or *tors*, 0 otherwise.

With just a vanilla feature approach, the model has some more room to improve. Some things to consider might have been to use a non-SGD approach or a dedicated library to find the weights such as pymc3 or pystan. pymc3 and pystan would have not only given the weights for each of the aforementioned features but also would have provided credible intervals on the weights themselves so as to better inform what features were truly noisy and/or statistically different from 0. Furthermore, in this method, nothing was done to stem the words (such as with NLTK) nor was anything done to mark stop words.

Hidden Markov Model Approach (Weitung Liao):

For this section, the code for this approach can be found here on Weitung's GitHub:
https://github.com/Ryo0929/NER-tagging-using-hidden-markov-model

Viterbi code implementation reference : https://en.wikipedia.org/wiki/Viterbi_algorithm

Preprocess :

1. Change any words that only occurs N times with "<unk>"

Parameter

1. Training data : Test data = 80 : 20
2. N=1

Input:

1. Observations
2. states = ("I", "O", "B")
3. Start_prob = {"I": 0, "O": 0.9457089, "B":0.0542911} // compute from training data
4. Transition_porb = //compute from training data

	I	O	B
I	0.606018	0.393982	0.00000
O	0.000000	0.952050	0.04795
B	0.579080	0.420920	0.00000

5. Emission_prob =
    a. Laplace smoothing : add Z to every words and Z*2 to total words when counting emission probability.
    b. Z=2

Other Change:

1. Using log when compute probability. : log(transition_prob) * log(emmision_prob)

```
##### Test result  #####
(4508, ' entities in gold standard.')
(4839, ' total entities found.')
(851, ' of which were correct.')
('\t', 'Precision: ', 0.17586278156643934, 'Recall: ', 0.18877551020408162, 'F1-measure: ',
0.18209051032416818)
####################
```

Findings:

When increase threshold of <unk>, F1 goes up but Recall goes down

Future Change:

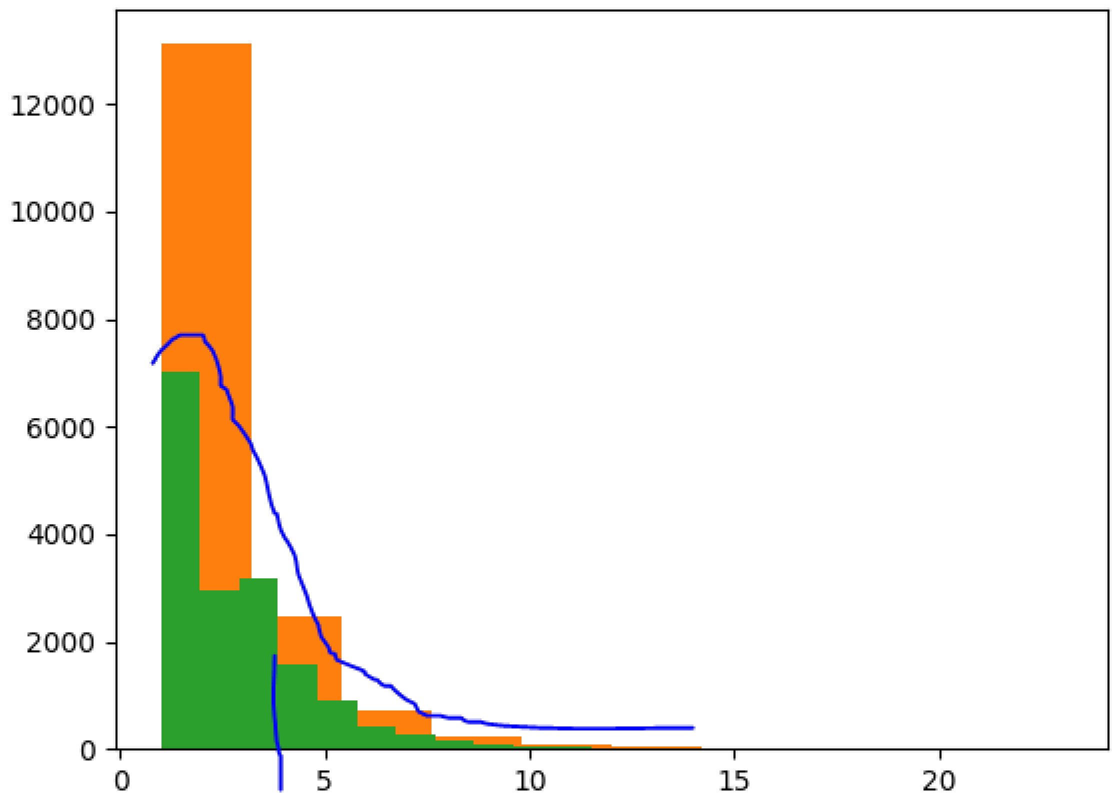
May try out Vanilla HMM

The Dumb-As-Bricks (DAB) Approach (Jay Seabrum):

For this section, the code for this approach can be found here on Jay's GitHub:
https://github.com/xjseabrum/nlp-files/tree/main/assgn3/dumb_method.py

This approach serves as the baseline to the other approaches. It makes 2 assumptions, which will be described further below.

In this approach, the average length of a tag and the distribution of the lengths of tags were found. The distribution of the tags looks like this:



Where the green histogram has the tags binned into bin widths of 1, the orange histogram with a bin width of 3, and the blue line represents a crude, hand-drawn approximation of the distribution. Note also that tag sizes less than or equal to three represent 80% of the tag lengths.

This distribution roughly resembles a Zipf distribution with equal density weighting on tag lengths 2 and 3. To account for this, a custom function, aptly named "stupid_dist()" in the python file mentioned above, roughly approximates the green histogram above.

This distribution is the first assumption that is held to be true for any validation set: the validation subset's tag distribution should roughly align to the larger set's distribution.

Now, the second assumption is as follows. The words were collected into a dictionary that contained the word itself and the proportion of the time that that word was a B tag. The dictionary was then truncated at the 80% level, meaning that only words that were tagged with a B 80% of the time remained in the random draws that happen later in the DAB approach. This dictionary was then labelled as “b_props”. Words that occurred fewer than 10 times in the dataset were masked with <UNK>, and the <UNK> masked words were found to have a B tag roughly 15% of the time.

The second assumption is that these words will remain highly likely to be begin tag words in the validation dataset.

Finally, a custom “predict” function was made that does two things:

1. Given a list of words, it checks to see whether that word exists in the b_props dictionary. If it does not exist, the word is labelled with an O.
2. If it does, a random number generator rolls a number between [0, 1). If that number is less than or equal to that word’s proportion in the b_props dictionary, then that word will be labelled as B. If the roll fails, the word is labelled with an O.
3. Upon a successful B label, the stupid_dist() function is called to assign the number of Is that should follow this B tag. This is checked against the number of words that are left in the respective sentence so as to not assign more Is than there are remaining words available.
4. This predict function repeats this process until the end of the word list. Note that the predict function *will not* reassign words with a label that have been assigned with a B or an I from the previous steps.

This approach is incredibly fast (roughly 30 seconds to run). However, with the DAB approach, the following are the metrics for model performance:

Precision	Recall	F1
0.2625	0.0405	0.0701

Incredibly, the precision is higher than would be expected. Recall and F1 being as bad as they are is more along the lines of what would be expected from this approach.

Even so, these numbers serve as the baseline to beat for the other approaches.