

Stacking of POS-Taggers

The objective of this project is simple enough that any layman gets it immediately: To write a program that classifies English words into parts of speech. The layman's understanding of part-of-speech (POS) classification belies the true complexity of the problem. My aim has been to try to tame this complexity by stacking three POS-taggers that each approach the problem from very different directions. I prepared a lot of data and successfully stacked the taggers. The models take a great deal of time to build and proper testing has still not finished, but I believe that the stacked POS classifier is better than any of its three constituents on their own.

THE TAGGERS

POS-tagging is an interesting enough problem that many people have attempted it already. The most well known attempt is the "Brill Tagger," developed by Eric Brill, who now works in the Natural Language Group at Microsoft:

<http://www.cs.jhu.edu/~brill/>

It is a rules-based tagger. Each word is given a POS somewhat naively, and then decisions on whether and how to alter the POS are made based on a long list of rules.

The second tagger chosen is called "Trigrams 'n' Tags." Instead of a list of rules, it "is an implementation of the Viterbi algorithm for second order Markov models," from the site:

<http://www.coli.uni-saarland.de/~thorsten/tnt/>

It was written by Thorsten Brants of the Universität des Saarlandes.

The third and final tagger is called, "MXPOST," uses the principle of Maximum Entropy to find part of speech. Written by Adwait Ratnaparkhi, of the Human Language Technologies Department at IBM, it can be obtained here:

<http://www.cogsci.ed.ac.uk/~jamesc/taggers/MXPOST.html>

Each of these taggers was treated as a black box. I wrote a Perl tool, '**posTool.pl**' that operates as a unifying wrapper to each tagger. It uses a perl library I wrote that provides one interface for all taggers. I can input normal English text, and this text gets reformatted and filtered so that it will be understood by whichever tagger is being used, and the output is also normalized so that it all looks the same. The only noticeable difference between the taggers is in their choice of tags.

THE DATA

The project required data. I decided to get as much as I could find. It was very hard to find, but when I finally found some, I found a great deal and I tried to use it all. I now see this as a mistake. While more data is *always* desirable, this amount was more than I had time to handle. I believe it was said once in our class that dealing with data often consumes the most time. The following is how the last month and a half has been spent:

- 5% searching for / selecting the taggers
- 10% coding a wrapper for the taggers
- 10% searching for data
- 5% sifting through and selecting usable data
- 67% cleaning data
- 1% finding weakness in each tagger
- 1% writing and optimizing stacking algorithm
- 1% writing this paper

That's 82% spent on managing data!

I found more than these sets of data (described below), but these were the ones that I believed would be the most usable. As a condition, they also had to be normalizable. Each set used its own syntax and one of several different tagsets. I chose the Penn Treebank (PT) set for the tagset, and the following format for the output:

Word|tag word|tag word|tag

I also made these restrictions that most datasets didn't have in their original form:

1. Each sentence on its own line
2. Each contraction, possessive, or other punctuation is united with the most appropriate word.
3. These punctuation marks can come at the beginning of a word: [\$ (' " -]
4. These marks can come in the middle of a word: [' -]
5. These marks can come at the end of a word: [, . : " ? !) %]
6. All punctuation comes on the word-side, not the tag side (i.e. Before the 'l' pipe)

For an example of #2, some datasets list "he's" as two words (he|tag1 's|tag2). These were united into "he's|tag1-tag2". That way the tag information is preserved, but the words appear as they would in normal text. Before text is passed to the taggers, possessives and common contractions are actually split up, but the tagged data is the reconcatenated before printing. As a side note, "NONE" was allowed for cases where the tag was unknown.

I chose the vertical pipe "|" instead of the commonly-used "/" because I wanted web addresses to be valid input material. The pipe is hardly ever used in common text. I also limited the allowed punctuation to the marks listed here in brackets: [

. , ' " - ? ! \$ % () / :]. The final tagger doesn't actually use most of these, but that decision wasn't made at this point, and a future version of the tagger might use them.

Here is an example of the regularization process. The following is a BROWN-Corpus sentence in its original format:

```
(S (PP-LOC (IN In)
  (NP (NP (PRP$-DEI your)
    (NNP Sept.)
    (CD 2)
    (NN page-one)
    (NN article))
  (` ` ` `)
  (VP (VBG Living) (PP-CLR (IN With) (NP#1002 (NNP AIDS))))))
(, ,)
(' ' ')
(NP-SBJ-PLE#0 (PRP it))
(VP (AUX was)
  (VP (VBN stated)
    (NP (-NONE- *-0))
    (SBAR (IN that)
      (S (NP-SBJ#49 (NNS patients))
        (VP (VBP hope)
          (SBAR (-NONE- 0)
            (S (NP-SBJ#1002 (PRP$#49 their) (NNS AIDS))
              (VP (MD will)
                (VP (VB become)
                  (` ` ` `)
                  (NP-PRD#1005 (DT a) (JJ manageable) (NN disease))
                  (PP (IN like) (NP#1006 (NN diabetes))))))))))))))
(. .)
```

The following is the regularized format:

```
In|NN your|PRP$ Sept.|NNP 2|CD page-one|NN article|NN "Living|VBG
With|IN AIDS,"|NNP it|PRP was|NONE stated|VBN that|IN patients|NNS
hope|VBP their|PRP$ AIDS|NNS will|MD become|VB "a|DT manageable|JJ
disease|NN like|IN diabetes."|NN
```

As you can see above, the final format is entirely different. Some things are ignored, like the tag “DEI,” or the tag “AUX,” which has no consistent mapping into PT, and some things are inferred and added in, like the ending double-quote.

The following is a description of the sets of data that were used. They were not obtained through normal channels. They were all found on random CD's in a closet at my workplace. I do not therefore have details on the provenance of each one, and I cannot be sure that any of them is a complete set. I also cannot confirm the extent to which they have been human-checked, though it is assumed they are, and the notes I found do seem to indicate this. Most can be obtained through the LDC (<http://www ldc.upenn.edu/>), but for a fee. Since it was not known how much overlap could be expected between the sets, I will later describe a protocol that I used for unquining each sentence.

ATIS: *A set of dialogs from speech data having to do with the airline industry*

This set is extremely small compared to the others, but useful because speech data is quite different from newspaper text. It is less structured and therefore could have some unique phrasing. The tagset is already PT, but the words are on separate lines. Here is a typical fragment:

```
=====
List/VB
[ the/DT flights/NNS ]
from/IN
[ Baltimore/NNP ]
to/TO Seattle/NNP
[ that/WDT stop/VBP ]
in/IN
[ Minneapolis/NNP ]
=====
Does/VBZ
[ this/DT flight/NN ]
serve/VB
[ dinner/NN ]
=====
[ I/PRP ]
need/VBP
[ a/DT flight/NN ]
to/TO Seattle/NNP leaving/VBG from/IN
[ Baltimore/NNP ]
making/VBG
[ a/DT stop/NN ]
```

in/IN

[Minneapolis/NNP]

Each sentence is bounded by lines. It is important to the unification of the datasets that sentences be separable. Each sentence can be compared and only one copy of each sentence kept. The tags used here are already Penn Treebank (PT).

BNC: *British National Corpus*

Only certain sections of this set were POS-tagged. Even so, This set is the largest. It also uses a larger tagset than the PT. It uses the original BNC codes, not the “Enriched” BNC codes. The mapping from this BNC tagset to PT was more complicated than what had to be done for the other tagsets, but it was doable with attention to detail. One example of these complications:

The BNC set has special tags for the verb “to be”:

VBB: some present tense forms of the verb be (e.g. am, are, 'm, 're, be (subjunctive or imperative), ai (as in ain't)).

VBZ: the -s form of the verb be (e.g. is, 's).

These (along with some other tags) map to the following two tags in PT:

VBP Verb, non-3rd person singular present

VBZ Verb, 3rd person singular present

Therefore, to map the BNC tag “VBB” to PT, you must check the word as well. If the word is “are,” for example, then the conversion is VBB -> VBP. If the word is “he's,” however, the conversion is VBB -> VBZ. “VBZ” maps without changing from BNC to PT, but there are other tag-name collisions with which more care must be taken, such as “PRP,” which in BNC marks prepositions, other than “of” (e.g. “about”, “at”, “in”, ...), while in PT it marks personal pronouns. Another complication with this set was all of the punctuation and special char codes, such as “Ë” for a capital E with dieresis, and “¼” for the symbol of the fraction “1/4.” All of these codes had to be replaced with something that wouldn't alter the POS landscape of a sentence. Here is an example of what the BNC data looks like:

<utt>

Guaranteed / AJ0 AJ0

Capital / NN1 NN1

Bond / NP0 NN1-NP0

<utt>

Stock / NN1-NP0 NP0

Market / NP0 NP0

Growth / NN1-NP0 NP0
 or / CJC CJC
 Your / DPS DPS
 Money / NN1 NN1
 Back / AVP AVP
 <utt>
 OFFER / VVB NN1
 CLOSES // NN2 VVZ
 29 / CRD CRD
 MAY / NP0 NP0
 1992 / CRD CRD
 <utt>
 All / DT0 DT0
 the / AT0 AT0
 attractions / NN2 NN2
 of / PRF PRF
 stock / NN1 NN1
 market / NN1-VVB NN1-VVB
 growth / NN1 NN1
 . / PUN PUN
 <utt>

BROWN CA-CR: *Brown Corpus, text from WSJ and other sources*

Very similar to ATIS, this first BROWN set is marked mostly with PT tags. There are a few non-PT tags such as “.” and “,”. These marks are just combined with the most appropriate word and their tags are dropped. Here is an example:

```

=====
[ It/PRP ]
has/VBZ been/VBN
[ my/PRP$ lot/NN ]
[ all/DT ]
through/IN
[ life/NN ]
to/TO associate/VB with/IN
[ eminent/JJ scientists/NNS ]
and/CC at/IN
[ times/NNS ]
to/TO discuss/VB with/IN
  
```

```

[ them/PRP ]
[ the/DT deepest/JJS ]
and/CC most/RBS vital/JJ of/IN
[ all/DT questions/NNS ]
./,
[ the/DT nature/NN ]
of/IN
[ the/DT hope/NN ]
of/IN
[ a/DT life/NN ]
beyond/IN
[ this/DT ]
./.
```

=====

BROWN 1987-1989: *Also from WSJ, but formatted differently and most likely containing a lot of non-overlapping data*

Most but not all words are on separate lines. They are parsed into clauses with the tag listed just before each word. Most of the non-PT tags here have to do with grammar structures, not the words themselves. Here is an example of the data:

```

(S1 (S (PP-LOC (IN In)
  (NP (NP (PRP$-DEI your)
    (NNP Sept.)
    (CD 2)
    (NN page-one)
    (NN article))
  (` ` ` `)
  (VP (VBG Living) (PP-CLR (IN With) (NP#1002 (NNP AIDS))))))
(, ,)
(' ' ' '))
(NP-SBJ-PL#0 (PRP it))
(VP (AUX was)
  (VP (VBN stated)
    (NP (-NONE- *-0))
    (SBAR (IN that)
      (S (NP-SBJ#49 (NNS patients))
```

```

(VP (VBP hope)
(SBAR (-NONE- 0)
(S (NP-SBJ#1002 (PRP$#49 their) (NNS AIDS))
(VP (MD will)
(VP (VB become)
(` ` ` `)
(NP-PRD#1005 (DT a) (JJ manageable) (MN disease))
(PP (IN like) (NP#1006 (NN diabetes))))))))))
(. .)

```

CRATER: *Corpus Resources and Terminology Extraction, a trilingual English, French, and Spanish dataset. (Only English was used)*

The format of this data almost looks like HTML. Word are on separate lines, listed along with other information. Each word as it “would appear” is between the “ORTH” tags. The POS is between the “CTAG” tags. This dataset is interesting for other types of projects as well because the corresponding lemmas are between the “BASE” tags. Sentences are marked by “S” tags.

This set uses the “Enriched BNC” tagset, which has more tags than the original BNC tagset. These new tags show a very fine granularity which can be seen in tags like these:

NNU1 : Singular unit-of-measurement noun (e.g. inch, liter, hectare)

ZZ2 : plural letter of the alphabet (e.g. A's, b's)

With such granularity it is relatively simple to convert these tags to PT, albeit time-consuming.

Here is an example from the data:

```

<S><TAB><TOK><ORTH>A</ORTH><BASE>a</BASE><CTAG>AT1</CTAG></TOK>
<TOK><ORTH>bi</ORTH><BASE>bi-directional</BASE><CTAG>JJ</CTAG></TOK>
<TOK><ORTH>-</ORTH><BASE>x</BASE><CTAG>x</CTAG></TOK>
<TOK><ORTH>directional</ORTH><BASE>x</BASE><CTAG>x</CTAG></TOK>
<TOK><ORTH>path</ORTH><BASE>path</BASE><CTAG>NN1</CTAG></TOK>
<TOK><ORTH>comprised</ORTH><BASE>comprise</BASE><CTAG>JJ</CTAG></TOK>
<TOK><ORTH>of</ORTH><BASE>of</BASE><CTAG>IO</CTAG></TOK>
<TOK><ORTH>an</ORTH><BASE>an</BASE><CTAG>AT1</CTAG></TOK>
<TOK><ORTH>input</ORTH><BASE>input</BASE><CTAG>NN1</CTAG></TOK>
<TOK><ORTH>connection</ORTH><BASE>connection</BASE><CTAG>NN1</CTAG></TOK>
<TOK><ORTH>and</ORTH><BASE>and</BASE><CTAG>CC</CTAG></TOK>
<TOK><ORTH>an</ORTH><BASE>an</BASE><CTAG>AT1</CTAG></TOK>
<TOK><ORTH>output</ORTH><BASE>output</BASE><CTAG>NN1</CTAG></TOK>

```

<TOK><ORTH>connection</ORTH><BASE>connection</BASE><CTAG>NN1</CTAG></TOK>
 <TOK><ORTH>,</ORTH><BASE>,</BASE><CTAG>,</CTAG></TOK>
 <TOK><ORTH>both</ORTH><BASE>both</BASE><CTAG>DB2</CTAG></TOK>
 <TOK><ORTH>having</ORTH><BASE>have</BASE><CTAG>VHG</CTAG></TOK>
 <TOK><ORTH>the</ORTH><BASE>the</BASE><CTAG>AT</CTAG></TOK>
 <TOK><ORTH>same</ORTH><BASE>same</BASE><CTAG>DA</CTAG></TOK>
 <TOK><ORTH>exchange</ORTH><BASE>exchange</BASE><CTAG>NN1</CTAG></TOK>
 <TOK><ORTH>interface</ORTH><BASE>interface</BASE><CTAG>NN1</CTAG></TOK>
 <TOK><ORTH>.</ORTH><BASE>.</BASE><CTAG>.</CTAG></TOK>
 </S>

PENN: Also from 1989 WSJ, but appeared to have data not contained in other sets.

Very similar to the BROWN 1987-1989: set. Here is an example:

```

( (S
  (NP-SBJ-1
    (NP
      (NP (NNP Georgia-Pacific) (NNP Corp.) (POS 's) )
      (JJ unsolicited)
      (ADJP
        (QP ($ $) (CD 3.19) (CD billion) )
        (-NONE- *U*) )
      (NN bid) )
    (PP (IN for)
      (NP (NNP Great) (NNP Northern) (NNP Nekoosa) (NNP Corp.) )))
  (VP (VBD was)
    (VP (VBN hailed)
      (NP (-NONE- *-1) )
      (PP (IN by)
        (NP-LGS (NNP Wall) (NNP Street) ))
      (PP (IN despite)
        (NP
          (NP (DT a) (JJ cool) (NN reception) )
          (PP (IN by)
            (NP-LGS (DT the) (NN target) (NN company) ))))))
    (. .) ))
  
```

SWBD: More speech-originated data, it was originally marked with “dysfluency”. All dysfluent phrases (marked with “NS”) were excluded. These sentences are parsed into clauses by brackets and braces. Some of the clauses are normal structural parts of sentences. Others are repeats, stutters, or any of the other things that show up in real speech but never show up in written text. The normalization of this set included the removal of phrase repeats, stutters, and many interjections. Here is a fragment from the dataset:

SpeakerA1/SYM ./.

Okay/UH ./ E_S

{F Um/UH ,/, } what/WP do/VBP you/PRP do/VB this/DT weekend/NN ?/. E_S

SpeakerB2/SYM ./.

{D Well/UH ,/, } {F uh/UH ,/, } pretty/RB much/RB spent/VBD most/JJS of/IN my/PRP\$ time/NN either/CC in/IN the/DT yard/NN or/CC at/IN nurseries/NNS buying/VBG stuff/NN for/IN the/DT yard/NN ./ E_S

SpeakerA3/SYM ./.

{F Huh/UH ./ } Which/WDT is/VBZ ,/, N_S

[wh-/XX ,/, d-/XX ,/, + what/WP do/VBP] you/PRP [plan/VB ,/, + have/VB planned/VBN] for/IN your/PRP\$ yard/NN ?/. E_S

SpeakerB4/SYM ./.

{D Well/UH ,/, } {F uh/UH ,/, } we/PRP just/RB bought/VBD our/PRP\$ house/NN ,/, {F uh/UH ,/, } last/JJ July/NNP ,/, E_S

{C and/CC } [it/PRP 's/BES ,/, + [[it/PRP ,/, + it/PRP ,/,] + the/DT house/NN] is/VBZ] forty/CD years/NNS old/JJ E_S

{C so/RB } [there/EX 's/BES ,/, + there/EX 's/BES] an/DT established/JJ lawn/NN and/CC whatnot/NN ,/, flower/NN beds/NNS ,/, things/NNS like/IN that/DT ,/, E_S

{C but/CC } [they/PRP 've/VBP ,/, + {F uh/UH ,/, } they/PRP 've/VBP] gone/VBN a/DT few/JJ years/NNS of/IN neglect/NN ./ E_S

SpeakerA5/SYM ./.

Uh-huh/UH ./ E_S

SpeakerB6/SYM ./.

{C So/RB ,/, } we/PRP 're/VBP in/IN the/DT process/NN of/IN ,/, {F uh/UH ,/, } revitalizing/VBG the/DT whole/JJ situation/NN ./ E_S

We/PRP 've/VBP got/VBN ,/, {F um/UH ,/, } different/JJ flower/NN beds/NNS in/IN front/NN and/CC ,/, {F uh/UH ,/, } [a/DT window/NN box/NN ,/, + a/DT built/VBN in/ RP window/NN box/NN ,/,] {F um/UH ,/, } trying/VBG to/TO get/VB some/DT color/NN back/RB in/IN those/DT ,/, get/VB the/DT trees/NNS trimmed/VBN up/RP ,/, {F uh/UH ,/, } get/VB rid/VBN of/IN a/DT few/JJ weed/NN problems/NNS and/CC things/NNS like/IN that/DT ./ E_S

SpeakerA11/SYM ./.

{F Oh/UH ,/, } okay/UH ./ E_S

Okay/UH ./ . E_S

My/PRP\$ wife/NN and/CC I/PRP ,/, we/PRP live/VBP in/IN Dallas/NNP too/RB ,/, E_S
{C so/RB ./ . } N_S

CLEANING THE DATA

A data cleaner was written for each dataset. This turned out to take up the largest portion of the time spent on this project. The code specific to this project is included in the appendix in order to give an idea of the complexities encountered.

First of all, a parser had to be written for each dataset. Besides all having their own formats, each set had its own long list of exceptions, and mistakes, of course. For the most part, these exceptions were regularized by a long list of exception parsing rules, and then parsed as any other line, but a very small percentage was simply excluded because it was too much time was required to regularize so many exceptions. This was especially true with the BNC set.

Linguistic data has many complexities, each of which had to be tackled one at a time. One difficulty was detecting word fragments and punctuation that should be put together, such as “he ’ll”, or “John ’s. Once regularized, they could be parsed like any other word. For the BNC set in particular, I found that word parsing had to be done very specifically, parsing each of these differently: newlines, numbers, number ranges, normal words, one-letter words, apostrophe extensions, starting quotes, ending quotes, alternate sentence endings, starting punctuation (e.g. “\$”, “(“). and punctuation without letters. Two states had to be monitored throughout each sentence:

1. IN / OUT of quotes
2. IN / OUT of parenthesis

This was needed in order to regularize data that had many mistakes and many nested quotes and parenthesis. For simplicity’s sake, nesting was not preserved in quotes or parenthesis, but these turned out to be unnecessary anyway.

I wrote a tool called **'posDatacleaner.pl'**, which takes the data type and file as inputs. It creates a “.cl” file in the same directory as the original file (cl for clean). The “.cl” files all have the standardized format discussed near the beginning.

MERGING THE DATA

Even without considering the possibility of overlap between these datasets, there is a possibility of multiple representation of identical sentences, albeit with possibly different tags (even if they were human-checked – we are human, after all). Now that the sentences have all been regularized, they have to be combined into one large set of unique sentences. It is

possible that multiple representations of more common grammar structures could actually help, but I think it is more likely that the same data coming from different sources would be overrepresented. The following protocol was used to “unique” the dataset:

1. Define a hash key with which to reference the sentence:
 - a. start off with the whole sentence as the key
 - b. make it lowercase
 - c. remove all punctuation
 - d. remove all numbers
2. The hash value is the sentence as it originally appeared
3. When there is a collision, proceed through the following steps for each word,tag combo:
 - a. unless words match (they could only differ in capitalization), keep word with capitalization switches
 - b. else keep words with lowercase
 - c. unless tags match, keep both tags, separated by hyphen
 - d. only keep one copy of each tag seen for a given word

The tool '**posDatacleaner**' was extended to include this functionality.

With this protocol, the dataset will be composed purely of unique sentences, but any sentences that come from multiple sources with different tags will contain the information of both. Using multiple tags to deal with ambiguity was already being done in some of the datasets. The following binary function will be used to determine how “correct” a tag is will do the following:

IF training data has tags X-Y on word W, and the classifier returns tags A-B
A-B will be considered a “match” to X-Y IF:
A=X or A=Y or B=X or B=Y

A similar cross-matching is done with higher numbers of tags. As long as at one tag matches, the tagging is deemed “correct.” It is hoped that matches that shouldn't have happened will be drowned out as low level noise. This function is the basis of the scoring function in '**posTool.pl**'.

I was not able at first to accomplish this cleaning on the entire merged dataset (6,750,151 sentences). No machine to which I had access could hold it all in memory, so I started by merging them into 200 large files (using '**posDatacleaner**'), a step that was not inherently necessary, but helped to simplify the matter. The next step was to make sure every sentence was unique without loading the entire thing into memory. I decided to put each sentence into a file that was named for its first two alphabet letters (lowercase). I had to remove the string “th” from the beginning of any sentence where it occurred (only for the purpose of file naming) just to make sure the “th” file wasn't abnormally

large. The rest of the distribution was spread more smoothly. There were many files that had too few lines to warrant their existence, so I merged all those files that had 1–500 sentences into one file, and then I merged all those files that had 501–2000 sentences into another file.

Then each of these files was processed so that it only had sentences that were unique by alphabet letters and spaces. Numbers, punctuation and capitalization were all discarded when computing uniqueness. Each file in itself was small enough to be loaded into memory. Now the entire dataset was only composed of unique sentences – 6,710,897 in all.

BUILDING A MODEL

The model first needs to be explained. Then the stacking algorithm can be described more easily. If the dataset is tagged by all three taggers, then for each word in each sentence we have four tags and the word itself:

M: output from the “Maximum Entropy” tagger
T: output from the “Trigrams 'n Tags” tagger
B: output from the “Brill” tagger
tag: The supervised answer from the human-checked data
word: The word that was tagged

There is a “specific” model and a “general” model. The “specific” model is specific to occurrences of a given word under certain conditions, where those conditions are the output of the three taggers. The “general” model is specific only to the output of the three taggers. The specific model is better because it tells you how that word was marked under similar conditions. If the word in question hasn't been observed, the back-off procedure is to look to the general model, which tells you which tagger to trust under the observed conditions, and after that to look to the most commonly observed tag considering those same conditions. The main idea is that the three respective outputs: M, T, and B, represent enough information about a given word, that reasonable improvement can be achieved even without considering n-grams. Besides, much n-gram information is inherent in the tagger outputs. It is possible that n-gram information might be useful for future versions of the stacker. This will be unclear until more experiments can be run. For now, these “conditions” of the observation (M, T, and B), will be expected to produce a gain.

'**posTool.pl**' was extended to build these models. The following is an example of how the specific model is built:

BUILDING THE SPECIFIC MODEL

1. INPUT FROM THE DATASET:

it|PRP was|NONE stated|VBN that|IN patients|NNS hope|VBP their|

```
PRP$ AIDS|NNS will|MD become|VB `a|DT manageable|JJ disease|NN like|
IN diabetes.`|NN
```

2. GET TAGGER OUTPUTS:

```
M:      it|PRP was|VBD stated|VBN that|IN patients|NNS hope|VBP
their|PRP$ AIDS|NNP will|MD become|VB `a|DT manageable|JJ disease|NN
like|IN diabetes.`|VBZ
```

```
T:      it|PRP was|VBD stated|VBN that|IN patients|NNS hope|VBP
their|PRP$ AIDS|NNP will|MD become|VB `a|DT manageable|JJ disease|NN
like|IN diabetes.`|NN
```

```
B:      it|PRP was|VBD stated|VBN that|IN patients|NNS hope|VBP
their|PRP$ AIDS|NNP will|MD become|VB `a|DT manageable|JJ disease|NN
like|IN diabetes.`|NN
```

3. DEFINE A LOCATION AND FILE UNDER THE MODEL DIRECTORY STRUCTURE:

```
dir = specific/M/T/B/   file = specific/M/T/B/word
```

Specifically, for the word "diabetes", the model file would be:

```
"model/specific/VBZ/NN/NN/diabetes".
```

4. ADD OBSERVATION TO FILE:

Add the present observation to the model file simply by adding the observed tag on a new line. Again, in the case of "diabetes" in the above sentence:

```
prompt# echo "NN" >> specific/VBZ/NN/NN/diabetes
```

Each time the word diabetes gets the three respective tag outputs VBZ, NN, and NN, the observed tag will be added to that file.

BUILDING THE GENERAL MODEL

The general model is built in a similar way, but will be used differently. The following is an example of how to build the general model using the same sentence:

1. INPUT FROM THE DATASET:

```
same as above
```

2. GET TAGGER OUTPUTS:

same as above

3. DEFINE A LOCATION AND FILE UNDER THE MODEL DIRECTORY STRUCTURE:

almost same as above: dir = "general/M/T" file = "general/M/T/B"

Specifically, for the word "diabetes," the model file would be "model/general/VBZ/NN/NN"

4. ADD OBSERVATION TO FILE:

Instead of adding the tag observed from the data, here I put the name of the tagger that was correct. From small early experiments I got the idea that the order of priority among the taggers should be (best to worst) should be M, T, B. In the case of the sentence above, M was incorrect (as defined by the observed data), so T is considered the tagger to trust under these conditions. Add the name of this tagger to the model file (In this part of the code, I used the following conventions: "me" = 'Maximum Entropy', "tnt" = 'Trigrams 'n Tags', "brill" = 'Brill') :

```
prompt# echo "tnt" >> general/VBZ/NN/NN
```

Every time this combination of tags is observed together, the "trusted" tagger is added to the general model file.

COMPACTIFYING THE MODEL

(Putting aside the debate over whether "compactify" is a real word) These model files are way too numerous and bulky to be used efficiently. A protocol will be used to compactify the specific and general models into much smaller files containing only essential information. 'posTool.pl' was extended to include this function of compactification.

HOW TO COMPACTIFY THE SPECIFIC MODEL

The essential information contained in the specific model is the knowledge of which tag is most common under the conditions represented by the output of the three taggers and the word itself. Therefore, a single file could be made for each word, and each line could list a tag-condition (i.e. The output of all three taggers separated by "_", e.g. "VBZ_NN_NN"), and the most commonly observed tag under those conditions (e.g. "NN").

The model is still building, but already, the original specific model file has 15 observations of the word "diabetes":

```
prompt# cat model/specific/VBZ/NN/NN/diabetes
NN
NN
NN
NN
NN...
```

(15 lines total, all the same)

Collecting data like this will ensure that if the word “diabetes” is seen in the future, and the three tagger outputs are VBZ, NN, and NN respectively, the stacker will feel confident about assigning “NN.” Now, to build the compact specific model, these steps must be taken on every specific model file:

1. READ ORIGINAL SPECIFIC MODEL FILE

Read each line in the original specific model file looking for two things:

- a. The most commonly observed tag
- b. The second most commonly observed tag and its relative incidence with respect to #1.

(The “conditions” are inherently known from the directory name where the file was found)

If the second most common tag is at least half as common as the first, then the final “authoritative” answer will be that the tag is a combination of the two. In the example of “VBZ/NN/NN/diabetes,” there is no second-most common tag, but if a word had “VBZ” as its most common tag, and “NN” as the second most common, and “NN” was observed at least half as many times as “VBZ”, then the final authoritative answer would be “VBZ-NN”.

2. DEFINE A LOCATION FOR THE COMPACT SPECIFIC MODEL FILE UNDER THE MODEL DIRECTORY STRUCTURE:

First, for the sake of not overloading directory listing functions, find the first two alphabet letters (e.g. “di”) in the word, and use them as a subdirectory. The compact specific model file for the word “diabetes” will be:

file = compact/di/diabetes

3. CREATE A KEY

This key will provide a marker for the line that we are about to add to the compact specific model file. This marker makes us able to find this answer at a later point when we observe these same conditions again. In this case, the key is “VBZ_NN_NN”.

4. ADD OBSERVATION TO FILE:

For “diabetes”, we would add a line containing our key and our authoritative answer for the given conditions:

```
prompt# echo "VBZ_NN_NN NN" >> compact/di/diabetes
```

In the future, when we again observe the word “diabetes”, and the respective outputs of the three taggers M, T, and B, are VBZ, NN, and NN, this stacker will give the answer “NN.” As you can imagine, even in situations where all three taggers are wrong, there is a chance of getting the right answer. It is at least arguable that there is enough information contained in the key and filename to represent a lot of the English language.

For example, the compact specific model file for “diabetes” in the model I am presently building has output from two sets of conditions so far:

```
prompt# cat model/compact/di/diabetes
VBZ_NN_NN      NN
NNS_NN_NN      NN
```

HOW TO COMPACTIFY THE GENERAL MODEL

The general model files are going to be compactified into one big file. The method is similar to the one above. It must be iterated upon every general model file.

1. READ ORIGINAL GENERAL MODEL FILE, FIND MOST COMMON OBSERVATION

Read each line in the model file looking for one thing: The most commonly trusted tagger under the given conditions represented by the outputs (and directory names) “M/T/B.” For above case, the original general model file is “general/VBZ/NN/NN”. It contains 540 observations. The observations are the names of the most trusted tagger in each situation. If the most common observation under these conditions is “tnt”, then our authoritative answer will simply be: “Trust the output T from the 'tnt' tagger”.

2. CREATE A KEY

same as above

3. ADD OBSERVATION TO FILE:

Here we print the key “VBZ_NN_NN” and the authoritative answer “tnt” into the compact general model file. There is only one of these, and it is called “model/compact/general”:

```
prompt# echo "VBZ_NN_NN      tnt" >> model/compact/general
```

USING THE MODEL

The tool 'posTool.pl' was extended to also use the stacker as a tagger. The interface to the stacker is the same as for the other three taggers. It is just as easy to use. Here's how it works:

1. INPUT TEXT TO TAGGERS

An input phrase is taken in the form of normal text. It is stripped of most punctuation. A few experiments were done on all three taggers and it was shown early on that the only mark that increases accuracy is the apostrophe. All other punctuation marks interfered somehow. Once the text (which can come from the command line or a file), is cleaned, it is split into sentences, and each sentence is passed one at a time to each of the three taggers.

2. ANALYZE TAGGER OUTPUTS

Each word of each sentence is then looked at one at a time. Knowing the output of each tagger, and the word itself, the compact specific model file is retrieved. If we were to come across the word “diabetes,” and the respective outputs of the three taggers M, T, and B, were VBZ, NN, and NN, the needed model file would be “compact/di/diabetes”, and a simple grep of our key “VBZ_NN_NN” on that file would reveal our authoritative answer, if the file exists. In this case it was observed, so the file does indeed exist:

```
prompt# grep VBZ_NN_NN compact/di/diabetes  
NN
```

A simple grep gives us this needed answer, but instead of using a grep, the whole file is read. This is also where the most “common” tag is found by a straight count. This tag is stored in case it is needed in step 4 below. This is all handled behind the scenes by 'postTool.pl'. This step is done for each word until the whole sentence is tagged with authoritative outputs.

3. BACK-OFF TO GENERAL MODEL

If step 2 doesn't produce an answer for a given word, the general model is used. In this case, a simple grep of our key on the compact general model file is all that is needed. If a grep of our key on that file reveals the answer “tnt”, then the stacker puts its trust in the output of the tnt tagger, which output “NN” for the word “diabetes” above. If these conditions were never observed (not likely) or if these conditions were observed, but all taggers were wrong (more likely), then there will be no line in the compact general file with our key. You will have to back-off to the “common” tag.

4. BACK-OFF TO COMMON TAG

In this case, use the most commonly observed tag from step 2 is used as the authoritative answer.

5. LAST-DITCH BACK-OFFS

There will be cases where there is no specific model file for a given word, and no key in the general file. In such cases, trust will be accorded first to “ME”, then to “TNT”, and finally to “Brill”. If none of them have an answer, “NONE” will be applied.

TESTING

The scope of this project is large and could not fully be explored in so short a time, but a good basis was laid for future improvements, and early experiments show that the stacker is providing useful answers. For example, using a twist on a well-known sentence, these are actual outputs of the three taggers, and then the last one is from the stacker (using a small model):

```
prompt# posTool.pl -t "the quick brown fox jumped all over the Indian tiger's  
second half-brother once removed" --tag me  
OUTPUT: the|DT quick|JJ brown|NN fox|NN jumped|VBD all|DT over|IN the|DT Indian|NN  
tiger's|NN-POS second|JJ half-brother|NN once|RB removed|VBD
```

```
prompt# posTool.pl -t "the quick brown fox jumped all over the Indian tiger's  
second half-brother once removed" --tag tnt  
OUTPUT: the|DT quick|JJ brown|JJ fox|NN jumped|VBD all|DT over|IN the|DT Indian|JJ  
tiger's|NN-POS second|JJ half-brother|NN once|RB removed|VBN
```

```
prompt# posTool.pl -t "the quick brown fox jumped all over the Indian tiger's  
second half-brother once removed" --tag brill  
OUTPUT: the|DT quick|JJ brown|JJ fox|NN jumped|VBD all|DT over|IN the|DT Indian|NN  
tiger's|NN-POS second|JJ half-brother|NN once|RB removed|VBD
```

```
prompt# posTool.pl -t "the quick brown fox jumped all over the Indian tiger's  
second half-brother once removed" --model model  
OUTPUT: the|DT quick|JJ brown|JJ fox|NN jumped|VBD all|DT over|IN the|DT Indian|JJ  
tiger's|NN-POS second|JJ half-brother|NN once|RB removed|VBN-VBD
```

Even though “ME” was originally the most trusted tagger, the stacker disagreed with ME’s choice on the words “brown,” and “Indian”, and rightly so. The stacker trusted its own models and found a better answer.

SMALL EXPERIMENTS:

A few experiments have been done to test the effectiveness of the methods laid out in this paper. Small ones were done first so that they could be completed quickly. Only a few days were required. The tagged sentences chosen for training and testing were chosen essentially at random from across the different original datasets and were not mixed (i.e. None of the training sentences were in the test sets). The number of sentences in each are listed below. Underneath are listed the total number of words in each test set, and the amount of words tagged correctly by each tagger, listing the Stacker first.

EXPERIMENT 1:

Training set: 1000 sentences

Test set: 200 sentences

Total Words	Stacker	ME	TnT	Brill
3452	3032	2922	2991	2959

EXPERIMENT 2:

Training set: 10000 sentences

Test set: 2000 sentences

Total Words	Stacker	ME	TnT	Brill
36927	33416	32249	32736	32457

EXPERIMENT 3:

Training set: 100000 sentences

Test set: 20000 sentences

Total Words	Stacker	ME	TnT	Brill
375943	347106	330227	334641	331584

What is most surprising to me is that the Maximum Entropy tagger comes in last each time. Tags'n'Trigrams seems to have the best single approach, but only slightly better than Brill. These numbers do show that taking the best of all three can create a tagger that functions slightly better than any of its constituents. An experiment on a larger model is described next.

LARGE EXPERIMENTS

Once a large model had been built, more meaningful tests could be done. The model took weeks to build. But first, some recap about the data: The datasets had been gathered into multiple directories with two-letter names. These two letters represent the first two letters of the text in the tagged sentences. The purpose was so that each list of sentences could be made unique without running out of memory. Besides the fact that the data is organized this way, I believe it is reasonable to assume that that these sentences are distributed randomly enough for testing the POS-taggers.

The only exceptions to the two-letter naming scheme were these two sets:

0-500

500-2000

These two sets are each a concatenation of many sets. When the sentences were originally distributed into these files, most of them were relatively short. All sets that had less than 500 lines were put into the first set "0-500", and all sets with between 500 and 2000 sentences were put into the second set "500-2000".

Here are the sizes (in sentences) of all the datasets:

36507	0-500	21866	DI	8003	GI	4089	KI	4992	PH	6916	VI
33995	500-2000	39508	DO	2752	GL	2018	KN	4878	PI	3573	VO
16862	AB	9992	DR	18835	GO	30900	LA	9669	PL	17461	WA
31505	AC	13363	DU	12210	GR	22008	LE	14988	PO	152542	WE
16346	AD	61515	EA	4253	GU	23189	LI	35964	PR	165602	WH
40164	AF	38275	EB	29967	HA	22721	LO	7708	PU	44444	WI
13650	AG	90565	EC	256962	HE	4296	LU	5599	QU	14936	WO
7711	AH	44501	ED	38711	HI	58533	MA	12334	RA	2187	WR
3402	AI	27122	EE	67113	HO	3447	MC	48686	RE	48681	YE
83903	AL	50994	EF	5475	HU	24656	ME	13442	RI	98877	YO
28076	AM	26299	EG	16372	IA	19593	MI	20595	RO		
225305	AN	18124	EH	5723	IB	2299	MM	5248	RU		
18012	AP	40650	EI	18180	IC	47758	MO	21880	SA		
24117	AR	6792	EJ	38093	ID	86837	MR	7733	SC		
99843	AS	4546	EK	2767	IE	3523	MS	38376	SE		
115797	AT	37790	EL	90923	IF	9963	MU	121515	SH		
5776	AU	51556	EM	6314	IG	18897	MY	31527	SI		
4451	AV	61038	EN	24970	IH	7871	NA	2568	SL		
5757	AW	29676	EO	3698	II	30998	NE	3662	SM		
3743	AY	64621	EP	3627	IJ	7276	NI	117699	SO		
19130	EA	5839	EQ	9824	IK	112233	NO	10154	SP		
47635	BE	155007	ER	17744	IL	2175	NU	36131	ST		
7133	BI	109925	ES	45077	IM	3490	OB	32579	SU		
5438	BL	43321	ET	276059	IN	2091	OC	2821	SW		
22351	BO	17093	EU	2238	IO	21276	OF	13470	TA		
17706	BR	43986	EV	3938	IP	27026	OH	16924	TE		
212982	BU	21413	EW	14153	IR	3287	OK	99341	TH		
23140	BY	13256	EX	147412	IS	3142	OL	6745	TI		
31618	CA	97542	EY	304519	IT	2236	OM	53353	TO		
7466	CE	10938	FA	2654	IU	94936	ON	18988	TR		
25305	CH	10257	FE	15196	IV	4805	OP	4161	TU		
4549	CI	31920	FI	37134	IW	14082	OR	15223	TW		
11503	CL	4307	FL	12761	JA	8545	OS	2012	TY		
68815	CO	80790	FO	5750	JE	15131	OT	34705	UN		
8237	CR	25498	FR	14345	JO	20441	OU	2890	UP		
7148	CU	10666	FU	18736	JU	7104	OV	21264	US		
15802	DA	5875	GA	4258	KA	22595	PA	3606	VA		
33709	DE	16691	GE	7092	KE	30109	PE	5128	VE		

The total number of sentences is 6,710,897. Most were used for training. A total of 101,252 sentences were used for testing. The following tests were left out during training and were used for testing:

sentences	/	set
13650		AG
8237		CR
5875		GA
3938		IP
7092		KE
19593		MI
3490		OB
4992		PH
16924		TE
17461		WA

As the large model built, there were various problems. The large size of the dataset made it prohibitively difficult to track down every build failure, and there were numerous ones. Most issues were resolved, but some of the data did not make it into the model for one reason or another. It would be reasonable to assume that more than 90% of the over six million sentences were incorporated into the model. None of them were used twice.

The results on the next page are listed in several columns, the first of which is the name of the dataset. Next is the total number of words. It is impossible for any of the taggers to reach 100% because some of the words in the dataset were tagged with "NONE". These were counted as words, but were never counted as a correct match. Next is the number of correct matches (in words) for the Stacker. Next is the percentage correct for the Stacker. After that come similar fields for the other three taggers: Max Entropy, Trigrams'n'Tags, and Brill:

SET	WORDS	STACKER		MAX ENT		TNT		BRILL	
AG.results	279788	266061	95.09%	250996	89.71%	252677	90.31%	250971	89.70%
CR.results	153143	143584	93.76%	134677	87.94%	136519	89.14%	135280	88.34%
GA.results	109612	103009	93.98%	96769	88.28%	98088	89.49%	97547	88.99%
IP.results	62151	59299	95.41%	53976	86.85%	54674	87.97%	53745	86.47%
KE.results	124589	117631	94.42%	110083	88.36%	111987	89.89%	111134	89.20%
MI.results	359630	337789	93.93%	315186	87.64%	320049	88.99%	317082	88.17%
OB.results	72352	68776	95.06%	64058	88.54%	64641	89.34%	64101	88.60%
PH.results	92402	86820	93.96%	81547	88.25%	82757	89.56%	81869	88.60%
TE.results	297573	279489	93.92%	261273	87.80%	264078	88.74%	262393	88.18%
WA.results	276989	260787	94.15%	240794	86.93%	247162	89.23%	245028	88.46%
TOTALS	1828229	1723245	94.26%	1609359	88.03%	1632632	89.30%	1619150	88.56%

The trend that was seen in the small experiments holds true for the large ones as well. The stacked tagger (Stacking of the other three taggers) consistently does better than any tagger on its own.

The order of accuracy among the taggers is also maintained. After the stacker, Trigrams'n'Tags next best. Then comes Brill, leaving Max Ent for last. This is definitely a surprise. Before any of the tests were done, I thought the order would be Max-Ent, Tnt, Brill. These taggers were treated as black boxes. They were not specifically trained against the data. It is possible that the order would have been different if they had been.

CONCLUSION

The stacking algorithm described in this paper is an effective way to combine the output of multiple POS-taggers and improve accuracy over a given tagger on its own. It's overall accuracy was 94.26%, a significant jump above the second place accuracy 89.30%. If pure accuracy is desired, this algorithm is excellent. The drawback is the amount of time required for model building. Despite this drawback, this method could find applications in many other fields.