Assignment One

Royi Rassin

Hidden Markov Model (HMM):

Handling unknown words (1):

Two approaches were taken. The first, training an *UNK* token, the second, implementing word-signatures.

Training *UNK*

While building the "emissions" vocabulary, words with frequency of one were also counted towards the *UNK* token.

Using *UNK*

If a word is not found in the "emissions" vocabulary, we use it instead.

Implementing Word Signatures:

The following word signatures were used – many of them were inspired from the Collins paper:
Plural suffixes: 's','es', 'ies, 'en', 'ies' – choosing plural suffixes is supposed to help the model differentiate between tags like NNP vs. NNPS, NN vs. NNS, and the likes.
Uppercase word (all-caps) – an uppercase word, can help us understand what type of tag is usually uppercase.
Lowercase word (no-caps) – regular lowercase word, can help us understand what type of tag is usually lowercase.
Capital letter and the rest lower – signifies the first word of a sentence, or proper nouns

Four-digit-number – can be a year.
Two-digit-number – can be a month or day.
Other numbers – any sort of number that is not two or four digit long.
'ed' suffix – signifies a past tense verb.
'ing' suffix – signifies an independent noun (morning, ceiling, reading – the reading of a will), can also signify a name (Reading Power Station), reading – verb. However, from my experience, it mostly indicates a verb, and that was the thinking behind this suffix.

Using Word Signatures:

If a combination of a word and a tag appeared four times (or less) in the emissions dictionary, we extracted the appropriate signatures for that word and computed the sum of their scores: the sum of the negative log of the frequency of the signature with the current tag, divided by the total frequency of that signature: ! −log ,34 (!"#$%&sig,t_i)*+,-(/01)().

For rare words, we combine between their emission-score and their word-signature with interpolation ($\lambda_4 = 0.7, \lambda_5 = 0.3$ respectively) .

Pruning Strategy in the Viterbi version (2):

The pruning strategy is straightforward: instead of going over all possible tags per loop, we go over all of the tags we have seen for a word during the training phase. So for instance, if we saw that the word ‘hardball’ was only seen with ‘NN’ during the training phase, then that will be the only tag we will check for that word.

Test Scores (3):

Newswire dataset (POS task):

HMM-Greedy: 93.3%
HMM-Viterbi: 95%
Feature-based-classifier: 96.2%

NER dataset (4):

HMM-Greedy:

Greedy-NER per-token-acc: 71.9%
NER-eval-acc: 94.5%

```
All-types  Prec:0.7975253093363329 Rec:0.715920565466173,
F1:0.7545228804540617
       ORG  Prec:0.6446969696969697 Rec:0.6346010439970171,
F1:0.639609169485156
       PER  Prec:0.886411149825784 Rec:0.6905537459283387,
F1:0.7763198046994202
       LOC  Prec:0.8630536130536131 Rec:0.8062057702776265,
F1:0.8336616943428089
      MISC  Prec:0.7531865585168018 Rec:0.7049891540130152,
F1:0.7282913165266106
```

HMM-VITERBI:

Viterbi-NER per-token-acc: 74.6%
NER-eval-acc: 95%

```
All-types  Prec:0.8076711323178565 Rec:0.7406597105351733,
F1:0.7727153015538583
       ORG  Prec:0.6762132060461417 Rec:0.633855331841909,
F1:0.6543494996150885
      MISC  Prec:0.808433734939759 Rec:0.7277657266811279,
F1:0.7659817351598174
       LOC  Prec:0.8445070422535211 Rec:0.8160043549265106,
F1:0.8300110741971207
       PER      Prec:0.8701953371140516 Rec:0.749728555917481,
F1:0.805482648002333
```

FEATURE-TAGGER:
FeatureTagger-NER per-token-acc: 86.15%
FeatureTagger-NER-eval-acc: 97.28%

```
All-types     Prec:0.8309535532578368 Rec:0.8520700100976102, F1:0.8413793103448275
      LOC     Prec:0.900054914881933 Rec:0.8922155688622755, F1:0.8961180973209404
      PER     Prec:0.8327383987761346 Rec:0.8865363735070575, F1:0.8587956876150408
      ORG     Prec:0.7231404958677686 Rec:0.7829977628635347, F1:0.7518796992481203
     MISC     Prec:0.8626309662398137 Rec:0.8036876355748374, F1:0.832116788321168
```

Difference between Greedy Tagger to Viterbi (5):

The obvious difference is that Greedy Tagger takes the greedy approach: the optimal **local** decision, whereas Viterbi is a dynamic-programming driven algorithm, and it can "look back" and find the optimal "path" of a tag sequence: y_4, y_5, y_6 ... , $y_,$. Consequently, Greedy Tagger is fast ~ 0.75 seconds, while Viterbi returns a solution in ~ 4.76 seconds. Another difference is that Viterbi needs a 3-dimensional data structure to store all of the possible paths of a sentence. We believe that while in a dataset such as this the memory and speed overhead is negligible, for a significantly larger data-set, one with longer sentences (increases space overhead. although, one can always free the used memory after generating the tag of every sentence) or many more sentences (increases time overhead) an increase of 1.7% in accuracy is not necessarily worthwhile (one has to take in account the machine that runs the algorithms).

FeatureTagger – batch prediction (bonus points):

Instead of calling 'predict' on every single word, we can simply call it on all of the words with the same index (i.e. all of the words with index 0 from all of the sentences). Since every sentence stands by itself, we are not relying on predictions of prior sentences, and we can batch-predict. We are down to 118 predictions (the longest sentence is 118 words long). Also, an option that we decided not to continue with, but was theoretically explored is implementing a predict function by ourselves and thus save time of redundant checks (that the data is 2D, number of features is correct and so on…), since we know exactly what is sent to our model. The idea for the custompredict method came from here: https://maxhalford.github.io/blog/speeding-up-sklearn-singlepredictions/ just to reiterate: we did not copy-paste or use any code from the internet! Regardless, it was an interesting read!

Difference in behavior between the datasets (6):

The POS-train has more than two times the samples the NER-train does. While the opposite is true for the dev sets. Furthermore, the NER dataset is broken down to documents, noted by a "DOCSTART-" when a new one begins – as a result, sentences between the "-DOCSTART-" symbols share some context. Conversely, the POS-dataset is broken down to sentences, where every sentence stands by itself. It is also worth mentioning that the tags in the NER dataset are encoded using BIO-scheme with the tags: PER, ORG, MISC, and LOC:
MISC – miscellaneous
LOC – location
ORG – organization
PER – person

BIO:
B – marks the beginning of a chunk
I – inside a chunk
Tags that don't belong to a chunk are tagged "O"

Many words' tag is "O", this is also the reason we do not include the correct prediction of "O" while calculating accuracy. Whereas the POS dataset is much more diverse in its tag-distribution (47 possible tags in POS-train vs 8 in NER-train).

Span scores are lower than accuracy scores in the NER dataset (7):

Accuracy is not sensitive to cases where there is an imbalanced representation of classes (such as the case in NER with "O"), in these cases, F1 is the better choice. If we do not exclude the correct classifications of "O" from our per-token-accuracy calculations, we will arrive to a very high score (95-97%). In fact, if we are to predict for every word that its label is "O", we will arrive to an accuracy of 83.3%. Obviously this does not reflect a classifier that learned anything, it is comparable to an all-yes classifier. For the same predictions and with the exception of the "O" class, the F1 score for each one of those classes will be zero, since the Recall and Precision will be zero too. And if we use F1 in its aggregate form, it will indeed reflect these lackluster results. As such, the accuracy score is obviously higher if we count the correct predictions of the

"O"s, whereas for the SPAN-metrics, the problem is addressed, since these metrics are suitable for an imbalanced dataset, where one class is ubiquitous and the others are not.

- We mention F1, but it also includes Recall and Precision since it is a combination of those two.
- This answer also covers: "how much lower is the NER acc from the POS? Look at the data and think why."

Modifying the HMMTagger (Viterbi) to improve accuracy on NER (8):

As per the suggestion in the assignment-document, we have modified Viterbi to go over the FeatureTagger's scores instead of the transmissions and emissions. In the algorithm itself, the only difference was vectorizing the words and sending them to the model's "predict_log_proba" function. Since the model's output is also a log of a number in [0, 1], no more changes were necessary. The results are only mildly better but with a much longer running-time and in our opinion, not worth the improvement.

NER-Accuracy of HMM-tagger compared to the feature-based tagger:

The per-token accuracy of the HMM-tagger (when not counting the 'O's) is 74.6%, while the feature-based's accuracy is 86.15%. A clear advantage the feature-based tagger has over HMM is more freedom over what the classifier learns, and specifically, the ability to incorporate external-knowledge such as gazetteers. A second up-side, is the ability to design features for words that are not rare (that is not to say you can't do that with Viterbi, but that's not how we implemented it). With a feature-based tagger we can utilize features that are not necessarily frequencies, unlike word-signatures (again, it is possible to do that with Viterbi, but that's not the classic way to implement it). As such, for the NER task, the feature-based tagger is a more suitable choice, and that is reflected in the accuracy results.

Improving the accuracy of the NER tagger (9):

On the first go we tried running the FeatureTagger with the features:
previous-tag 3-len-suffix
form

Running into converging-errors (no convergence), we increased the max_iter to 5000, and the problem ceased. It is safe to assume the actual number of iterations is significantly lower than this.

The first three features resulted in somewhere around 73% per-token-accuracy (no "O"s). From there, we added:

3-len-prefix – a prefix of len 3 of the word
previous-previous-tag – the concatenation of the previous two predicted tags
is_first_word – is this the first word in a sentence? is_last_word – is this the
last word in a sentence? pw – previous word (for current index i, pw is i-1)
ppw – previous previous word (for current index i, pw is i-2) nw – next
word
nnw – next next word

Again, we have a decent increase, to the 76% ± range.
Surprisingly, removing "previous-tag" and "previous-previous-tag" increased the results by 2%. Some more experimentation and going through the attached paper in the assignment-doc lead us to add more Boolean features: does the word contain a digit? Is it an upper-case word? Does the word begin with a capital letter and the rest is lower? Does the word have some sort of punctuation in it? With the exception of the latter, incorporating them increased the results to 84.9%. Finally, adding gazetteers from the GitHub-link in the assignment-doc boosted the results to 86.15% by incorporating three features: is_name – is the word in the "first_name" or "last_name" gazetteers? The feature: is_location – in a similar fashion to is_name, and is_organization, which is made up from the gazetteers: sports-league, business-brand, businesssponsor, cvg-developer, education-university, government-agency, sports-team, and tv-network. It is worth mentioning that modifying Viterbi to work with the feature-based scores increased the results ever-so-slightly.

In summary:

<u>Didn't work:</u> adding too many features – it probably made the space too sparse, or features that are not that useful for the data we are dealing with – is some sort of punctuation present in the word?

<u>Worked:</u> removing previous-tags as a feature, checking whether a word starts or finishes a sentence, gazetteers, reading through "Design Challenges and Misconceptions in Named Entity Recognition" by Lev Ratinov and Dan Roth.

<u>Model includes:</u>

<u>Features:</u> 3-len-suffix, 3-len-prefix, word-form, is_upper, is_last, is_first, is_contain_digit, is_first_word_or_name, previous-word, previous-previous-word, next-word, next-next-word.

<u>Gazetteers</u> – first_name, last_name, location, sports-league, business-brand, business-sponsor, cvg-developer, education-university, government-agency, sports-team, and tv-network.

Logistic Regression with max_iter=5000 – possibly much lower than this, since no overfitting has occurred.