Universitat de Barcelona

Natural Language Processing

Project 2: Name Entity Recognition (NER)

Irene Bonaforte, Otis Carpay, Vladislav Nikolov, Marcos Plaza, and Marc Taberner

June 20, 2022

UNIVERSITAT DE BARCELONA

1 The Structured Perceptron

1.1 The Perceptron

The first notions of the perceptron were presented by McCulloch and Pitts in 1943 [3]. It was, however, not until 1958 when Rosenblatt [6] at the Cornell Aeronautical Laboratory that build the Mark 1 Perceptron Machine, with the first implementation of the Perceptron Algorithm. Given an n-dimensional input vector, the perceptron learns a weighted sum of the given input components to consequently pass it through an activation function which will discriminate between two classes, making therefore, the algorithm a binary classifier. Originally, the model used a step function as the activation function, thus presenting the representation of figure 1.

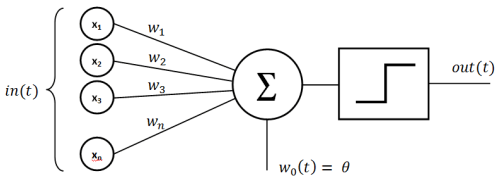

Figure 1: Perceptron Model

Let $D = ((x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n))$ be the sequence of training examples, where $x_i \in \mathbb{R}^n$ and $y_i \in \{-1, 1\}$ for all i. That is, we discriminate between two distinct labels, label 1 and label -1. Then the perceptron algorithm is the following:

– Initialise $w_0 = 0 \in \mathbb{R}^n$
– For $(x_i, y_i) \in D$:
– Let $y^{'} = sgn(w_t^T x_i)$
– If $y^i \neq y^{'}$
– Update $w_{t+1} = w_t + r(y_i x_i)$

This simple algorithm yields very strong results that ensure convergence whenever the data is linearly separable under very weak assumptions[5]. Nonetheless, the limitations of the algorithms are clear, there is not insurance of how well the data is split, nor any particularly interesting result for the non-separable case. Therefore, as the years passed, more and more algorithms and models were based on Rosenblatt work, expanding his ideas, e.g. by introducing the idea of kernels, adding more neurons that interact with each other, adapting the algorithm to data with a certain structure, or enriching the training samples the algorithm could work with.

1.2 Discriminative Sequence Labelling and the Structured Perceptron

Let's assume that a sequence of observations is given, with each individual observation coming from a finite set Σ. We would want to determine out of the possible elements of the set Λ, set of labels, that sequence of labels that suits the observations the most. An approach to tackle this problem could be to adopt a generative approach, estimating the joint probabilities of observations and labels, maximising, then, in the label space. The joint distribution can be estimated using distinct models, for example a Hidden Markov Model (HMM) [1].

Generative approaches such as HMM have notable limitations, the most noteworthy of which is the impossibility to add arbitrarily defined features to the model. The previously defined perceptron works as a linear classifier that accepts any feature, given that is numerical. Nonetheless, it works as a binary classifier and hence is unable to segregate between a set of labels Λ of more than two elements. Moreover, the sequential structure of the data is lost, as the perceptron does not work with state dependencies in any way. The algorithm needs to be adapted.

Let's assume that our label set Λ is larger than 2. Then, to solve the classification problem it suffices to define one neuron, one perceptron, for each of the classes, and then segregate between classes given a condition on the outputs of each individual perceptron. For instance, finding the weights that maximise the weighted sum for all corresponding labels, as depicted in figure 2. In other words, assume that we have a set of weights for each of the labels in $\Lambda = (1, 2, \cdots, |\Lambda|)$, $W = \left(W_1, W_2, \cdots, W_{|\Lambda|}\right)$. Our class prediction formula could be:

$$\hat{y} = argmax_{y \in \Lambda} W_y^T X$$

Naturally, we generally have a training set for which we can train the weights for each label. Assume that we have $D = ((X_1, y_1), (X_2, y_2), \cdots, (X_n, y_n))$ training set, and name E the number of epochs to train our dataset and r

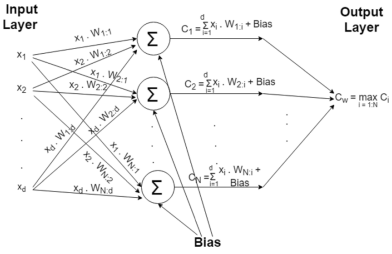

Figure 2: Multi-Class Perceptron Model

the learning rate. Then the equivalent formulation for the multi-class perceptron would be:

```
– Initialize W_i = 0 ∀i ∈ [1, 2, ··· , |Λ|]
– for i from 1 to E:
–   for X_i, y_i ∈ D:
–     y' = argmax_y W_y^T X_i
–     if y' ≠ y_i:
–       W_{y_i} = W_{y_i} + rX_i
–       W_{y'} = W_{y'} − rX_i
```

This newly presented perceptron algorithm is able to work with multiple labels. Nonetheless, it is still unable to work with structured data. It was not until 2002 that a paper published by collins[2] presented the structured perceptron. An architecture almost identical to that of the perceptron, the model was able to treat structured data. The idea is to combine the perceptron as linear classifier with an inference algorithm. Classically the Viterbi algorithm is chosen. The original structured perceptron is based on a linearization of the HMM, although it's not limited to it.

Assume that we are given X and Y sequences of items and labels respectively. Define a joint feature function $\phi(x, y)$ that maps a training sequence x and its associated labels y to an n-dimensional space. Assume as well that we are given a function "Gen" that generates candidates predictions given a training sequence. Then if E is the number of epochs, D the dataset, and r the learning rate, the Structured Perceptron Algorithm is the following:

```
– Initialize W = 0
– for i from 1 to E:
–   for X_i, Y_i ∈ D:
–     Y' = argmax_{Y∈Gen(X_i)} W^T φ(X_i, Y)
–     if Y' ≠ Y_i:
–       W = W + rφ(X_i, Y_i) − rφ(X_i, Y')
```

Note that in this pseudo-code, W was treated as a matrix of weights, rather than having a weight associated to each label, it does not affect in any way the algorithm. Here, the resemblance with the multi-class perceptron is clear. A joint feature of both sequence of data and labels is the main difference. This map is able to maintain the sequential structure of the data. Moreover, the features can be arbitrarily defined. Hence, this newly Structured Perceptron solves the two problems we had. Furthermore, a function that generates candidates is also provided. Normally a Viterbi algorithm works as a adequate "Gen" function, although others might work perfectly fine too.

1.3 Structured Perceptron and NLP

The Structued Perceptron is defined, therefore, for those problems where the structure of the data is relevant. An obvious example is language models, in which data is presented by a sequential structure. Data is presented as sentences consisting of ordered elements, which are words.

One can take logarithms in the HMM model, and will discover that it is actually a linear model. The embedding, then, is the one obtained combining the probabilities defined in the HMM, one can find more information in [2]. In this case one can work with features that already present in the HMM, that is, treat the embedding as the distinct HMM features, the initial, transition, emission and final features. But additionally, due to the flexibility of the embedding, many other features can be included, taking into account, for example, prefixes and suffixes of words, digits, roots of words...

This linearitzation of the HMM, and Structured Perceptrons, in general, have alternative benefits to those mentioned before. One notable, could be the possibility to freely and "carelessly" work with non-appearing words in the training when testing the data, that is caused since the word is not directly stored, but rather one directly works with the associated labels.

All these benefits, although originally thought for the Language Model case, the extrapolation to alternative problems that require the Structured Perceptron to be solved, is trivial, only the features need to be adequately adjusted.

2 Structured Perceptron for NER

The present report focuses on the application of the Structured Perceptron for the Named Entity Recognition (NER) problem. The goal is to locate and classify named entities in an unstructured test. We worked with a dataset consisting of sentences where each word is tagged with a label according to the sort of named entity. The 17 possible labels are 'O' (not a named entity) and various types (geographical entity, organization, person, geopolitical entity, time indicator, artifact, event, and natural phenomenon), either as first word (prefixed 'B' for beginning) or later word (prefixed 'I' for inside) of the specific entity. The training set vocabulary consists of 31 979 words and the test set vocabulary of 43 955. Hence, there are quite some new terms and the model will have to rely on other features to recognize the named entities of these unknown words.

2.1 HMM-like features

Using the data provided and working with the skseq library, and their built-in StructuredPerceptron function, one could fit the data to a Structured Perceptron with only HMM-like features (i.e. features representing initial scores per tag, emission scores per tag-word pair, transition scores per tag-tag pair and stop scores per tag). This model uses 39802 parameters in total.

2.2 Richer Feature Set

An important limitation of HMM is that they do not allow to incorporate arbitrarily defined features. The Structured Perceptron allows us to model the state dependencies just as with HMM, while being able to incorporate additional features with any kind of domain knowledge. The only requirement for this is that we can map each feature to an edge or node in the Viterbi trellis. This is, features should be a function of $\phi(y_i, y_{i-1}, X, i)$. To capture information that facilitates distinguishing the tags in the class at hand, inspired by classical reviews on NER and NER features ([4] and [7]), we have implemented the following 229326 features:

2.2.1 Word structure

We define a set of features that fire when detecting word shapes that we consider useful for identifying our objective tags.

- **Acronyms:** useful for detecting locations and organizations.
- **Is digit:** fires for numbers.
- **Number of digits:** to make it easier to capture months and years in the time tag, we have added features for tokens consisting on 1, 2 and 4 digits.
- **Digits and suffix:** for similar reasons, we have made features capturing digits followed by "st" (1st), "nd" (2nd), "rd" (3rd), "th" (4th) and "s" (70s).
- **Digits with floating points.**
- **Word with dots.**
- **Word with hypends.**
- **Word with apostrophe.**
- **Quoted words.**
- **Uppercases:** we have added features firing for words starting with uppercase, words where all the letters are uppercased and words with some uppercases mixed in the names.
- **Word length:** we have added features firing at different common word lengths (1,2,3,4,5,6,7,8,9,10,10-15,$\geq$ 15)

2.2.2 Local knowledge

HMM-like features and word structure features take only into account the current word information. However, one of the advantages that our model has to offer is that the feature vector can depend on the whole X sequence. This is particularly useful for the NER problem, as many of the words we are interested are in non-unit named-entity chunks (e.g. hurricane Katrina or United States of America), and difficult to model using only local information. To benefit from this possibility, we add a set of features that are similar to emission scores but with previous word and following word. The conditions required to trigger the feature are $[w_{i+1} = a, y_i = b]$, for next word, and $[w_{i-1} = a, y_i = b]$ for previous word. Additionally, we notice that many of the words in non-unit named-entity chunks are capitalised. Therefore, when a word is capitalised, we include features not only for the previous and next words, but for the three previous and the three next words. Notice that this leads to a very important increase in the dimensionality of our feature space.

2.2.3 External knowledge

2.2.4 Suffixes and preffixes

Many of the words we want to tag are characterized by suffixes and preffixes. For instance, most words with the GPE tag, representing nationalities, end in "-an" (e.g. american). To identify this, we have downloaded a list of most common suffixes and preffixes in the English language, and created features per each state and affix combination that fire when the affix is detected.

2.2.5 Word Clusters

Another possible way of including external knowledge is to use trained vectorizers. We have used Word2Vec to generate 300 dimension embeddings representing our words. To incorporate this information into our model, we have divided the words in 100 clusters of words using this embeddings, and added a feature for each cluster-tag combination that is triggered when the current word belongs to the cluster.

2.2.6 Further work

Other useful external information that we have not added due to time constraints are grammatical tools (i.e. lemmatization and stemmers, and Part Of Speech detection, which can be very helpful for this task). Additionally, gazetteers are a popular way to include domain knowledge. These are manually curated lists with well known people, popular events, locations, etc. An interesting idea is to use the wikipedia article titles and its associated categories as external information of the possible categories of our words.

3 Results

3.1 Structured Perceptron with HMM-like features

In the following section we present the results for our baseline model, the Structured Perceptron using only HMM-like features. Excluding the 'O' tag, we obtain the following performance:

	Accuracy	Weighted F1 Score
Training	0.806	0.840
Testing	0.230	0.354

In which we can observe how the training Accuracy and F1 Score, far outweigh that of the testing set, implying that the model is not completely able to generalize with the data given.

As well, we can observe the results with the tag 'O' included, naturally, far better results are obtained.

	Accuracy	Weighted F1 Score
Training	0.963	0.963
Testing	0.876	0.847

Comparing both of the metrics with and without the 'O' tag, one might deduce that most likely the majority of the miss-tagged data is tagged as 'O', below lies the confusion matrix which confirms our first assumptions. Note that the larger and bluer a dot is, the larger the percentage of words with that predicted tag.

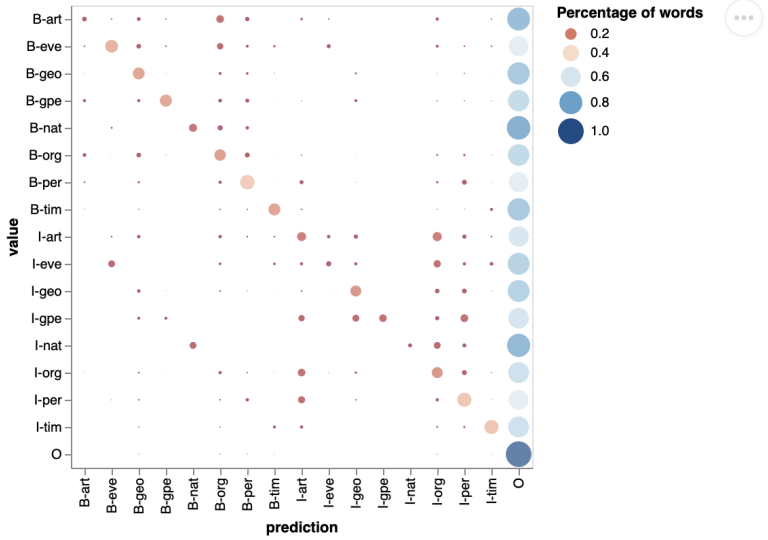

Figure 3: Confusion Matrix for Baseline Model

From this confusion matrix one can observe how the principal diagonal have larger dots than most of the rest of the tags, implying that the model comprehends the task in hand, nonetheless, a large majority is concentrated in the 'O' tag, the model is not capable enough to discern between the actual tag, and the 'O' tag. To further study this relations, we present a couple of examples of the "Tiny Test", to observe in which cases the algorithm is able to classify correctly the words, and in which cases is unable to.

Sentence	"Jack London went to Parris."
Prediction	'Jack/B-per London/B-geo went/O to/O Parris/O ./O '
Sentence	"The king of Saudi Arabia wanted total control."
Prediction	'The/O king/O of/O Saudi/B-geo Arabia/I-geo wanted/O total/O control/O ./O '
Sentence	"Apple is a great company."
Prediction	'Apple/O is/O a/O great/O company/O ./O '

3.2 Structured Perceptron with rich features

Let us now observe the results obtained by the Structured Perceptron after using the rich features that we have designed. The accuarcies and weighted F1 scores for the training and test sets excluding the 'O' tag are the following:

	Accuracy	Weighted F1 Score
Training	0.830	0.848
Testing	0.478	0.490

As we can observe, these results are much better than the ones that we previously had. Thanks to the features that we have crafted, the Perceptron has been able to generalize better than the baseline model. Although it is far from being perfect, the results are not that bad considering how difficult this task is.

Just like in the previous case, the results including the 'O' tag are far better as we can see right below:

	Accuracy	Weighted F1 Score
Training	0.962	0.963
Testing	0.894	0.899

Overall, the results have improved quite drastically, but there are still many words incorrectly tagged. In order to see how the model is performing, let us check the confusion matrix:

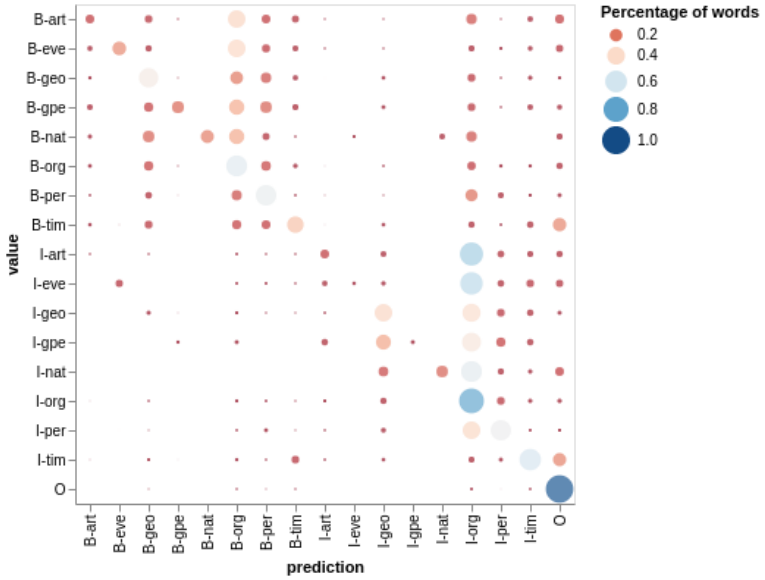

Figure 4: Confusion Matrix for Baseline Model

Compared to the previous confusion matrix, we can observe that now the Perceptron is not classifying almost all of the words as 'O', which is a huge step in the right direction. However, it seems that the model is now classifying more words as organisations ('B-Org' and 'I-Org' tags). Something similar happens with geographical localizations ('B-Geo' and 'I-Geo' tags), although it is not as frequent as with the organisations. Perhaps there are too many features generated for these classes and they need to be tuned down.

Finally, let us now run this new version of the Perceptron on the "Tiny Test" dataset and let's see what results we obtain this time:

Sentence	"Jack London went to Parris."
Prediction	'Jack/B-per London/B-geo went/O to/O Parris/B-per ./O '
Sentence	"The king of Saudi Arabia wanted total control."
Prediction	'The/O king/O of/O Saudi/B-org Arabia/I-org wanted/O total/O control/O ./O '
Sentence	"Apple is a great company."
Prediction	'Apple/B-per is/O a/O great/O company/O ./O '

As we can see, the model has made some mistakes, like considering Arabia Saudi as an organisation, Parris as a person and Apple as a person. However, in this case it has successfully detected Apple as an entity, which did not happen in the previous case.

4 Appendix: Extra exercise

In this part we report our efforts to improve the provided code.

To optimize code efficiently, we should find out where the interpreter spends the most time. First, we notice that the training (unsurprisingly) takes the longest, so we potentially gain the most by optimizing this part. To that end, we profiled a single-epoch run of the `fit` function of the `StructuredPerceptron` class using the `%%prun` cell magic. The truncated output is listed below.

```
    ncalls  tottime  percall  cumtime  percall filename:lineno(function)
         1    0.000    0.000  518.480  518.480 {built-in method builtins.exec}
         1    0.000    0.000  518.480  518.480 <string>:1(<module>)
         1    0.000    0.000  518.480  518.480 structured_perceptron.py:25(fit)}
         1    0.115    0.115  518.479  518.479 structured_perceptron.py:58(fit_epoch)
     38366    4.411    0.000  518.364    0.014 structured_perceptron.py:95(perceptron_update)
     38366    0.224    0.000  513.243    0.013 sequence_classifier.py:124(viterbi_decode)
     38366  164.527    0.004  306.158    0.008 discriminative_sequence_classifier.py:25(compute_scores)
     38366   69.183    0.002  206.636    0.005 sequence_classification_decoder.py:83(run_viterbi)
231708035  114.319    0.000  130.703    0.000 id_feature.py:113(get_transition_features)
 27456818   15.912    0.000  114.814    0.000 {built-in method numpy.core._multiarray_umath.impleme...
 13651677    8.504    0.000   84.610    0.000 <__array_function__ internals>:177(amax)
 13651677   12.457    0.000   66.556    0.000 fromnumeric.py:2675(amax)
 13651677   16.512    0.000   54.099    0.000 fromnumeric.py:69(_wrapreduction)
```

```
 13651677    9.243    0.000   51.338    0.000 <__array_function__ internals>:177(argmax)
 13651677   11.253    0.000   32.260    0.000 fromnumeric.py:1127(argmax)
 13651677   29.712    0.000   29.712    0.000 {method 'reduce' of 'numpy.ufunc' objects}
 13651677    7.869    0.000   21.007    0.000 fromnumeric.py:51(_wrapfunc)
231938524   16.432    0.000   16.432    0.000 {built-in method builtins.len}
 14443315    8.853    0.000   10.930    0.000 id_feature.py:98(get_emission_features)
```

Computation is mostly done by iteration of the `perceptron_update` function. Within this function, we find that the most time is spent in the `compute_scores` and `run_viterbi` functions. Since its optimization is somewhat simpler, we start with `run_viterbi`, defined in `SequenceClassificationDecoder`.

4.1 run_viterbi

This function can be optimized dramatically with a relatively small and obvious change. We replace the inner `for` loop in the Viterbi iterations with NumPy logic as follows:

```
# Viterbi loop.
for pos in range(1, length):
    for current_state in range(num_states):
        viterbi_scores[pos, current_state] = \
            np.max(viterbi_scores[pos-1, :] + transition_scores[pos-1, current_state, :])
        ...
```

Optimized:

```
# Viterbi loop.
for pos in range(1, length):
    viterbi_scores[pos] = \
        np.max(viterbi_scores[pos-1] + transition_scores[pos-1], axis=1)
    ...
```

Using `timeit` we find that the original function runs at 3.3 ± 0.1 ms, whereas the improved version runs at 344 ± 16 µs per loop, or roughly 10 times faster! Regardless of whether any further speedups should be possible, they will have little effect as the function now only occupies a small part of the execution time in `StructuredPerceptron.fit`.

4.2 compute_scores

Since the weight in the `perceptron_update` function has now shifted heavily to `compute_scores`, this is the logical next candidate. This function does not let itself optimize as easily as `run_viterbi`, so it is time to play the Cython card. We used the `%%cython` cell magic with the `-a` option to facilitate the optimization process. This option outputs the code with less optimized lines seeing a more yellow background. This makes it easy to see the effect of specific changes. We add the following type definitions at the top of the function of the variables used in the rest of the code:

```
def compute_scores(sp, sequence):
    cdef int num_states = sp.get_num_states()
    cdef int length = len(sequence.x)
    cdef double[:,:] emission_scores = np.zeros([length, num_states])
    cdef double[:] initial_scores = np.zeros(num_states)
    cdef double[:,:,:] transition_scores = np.zeros([length-1, num_states, num_states])
    cdef double[:] final_scores = np.zeros(num_states)

    cdef double score
    cdef int feat_id, pos, tag_id, prev_tag_id
    cdef double[:] parameters = sp.parameters
    ...
```

This utilizes the performant Cython memoryview interface into NumPy arrays. The memoryview objects are coerced back into NumPy arrays in the `return` statement of the function by calling `np.asarray`.

We experimented with further improvements by changing the output of the feature retrieval functions, but these introduced much more complexion than they reduced execution time. The cythonized code was pulled into a separate `.pyx` file and called using `pyximport` into `discriminative_sequence_classifier.py`.

With the presented changes we reduced the execution time from 5.33 ± 0.26 ms to 2.83 ± 0.15 ms per loop.

4.2.1 `get_transition_features`

Now that the calculations inside the `compute_scores` function itself have mostly been optimized, we find the next candidate by looking again at the above `prun` output. The original `compute_scores` function spends more than a third of its time (or more than half in the improved version) on the `get_transition_features` function defined in the `IDFeatures` class. Of course this is unsurprising as it is called in a triple `for` loop over $x \times \Lambda \times \Lambda$.

The function retrieves the feature id from either the `edge_feature_cache` or the `add_transition_features` function. The profiling output tells us that in the vast majority of instances the id is retrieved from the cache. The cache is a double python dictionary with the tags as keys. Unfortunately, we have not been able to optimize this. We attempted to replace the dictionary with a 2-dimensional NumPy array where the two tags form the index (the `add_transition_features` function was changed to return `feat_id` instead of `[feat_id]` as there is never more than one entry in the list). We expected the array access to outperform python dictionary access, but the adapted function took at least twice as long.

4.3 Result

Listed below is the truncated `prun` output of the `fit` function with the discussed improvements applied.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	262.630	262.630	{built-in method builtins.exec}
1	0.000	0.000	262.630	262.630	<string>:1(<module>)
1	0.001	0.001	262.630	262.630	structured_perceptron.py:25(fit)
1	0.129	0.129	262.629	262.629	structured_perceptron.py:58(fit_epoch)
38366	4.617	0.000	262.500	0.007	structured_perceptron.py:95(perceptron_update)
38366	0.276	0.000	257.180	0.007	sequence_classifier.py:124(viterbi_decode)
38366	0.080	0.000	231.091	0.006	discriminative_sequence_classifier.py:27(compute_sco...
38366	91.706	0.002	231.010	0.006	{skseq.sequences.cython.compute_scores.compute_scores}
231708025	110.695	0.000	128.471	0.000	id_feature.py:113(get_transition_features)
38366	11.589	0.000	25.605	0.001	sequence_classification_decoder.py:84(run_viterbi)

The profiling output of the inference step using `viterbi_decode_corpus` is similar (both rely on `viterbi_decode`) and the execution time is reduced by the improvements from 489 seconds to 224. Both steps are successfully optimized to take half as long.

Looking further at the output, we find that all of the options for improvement visible have been exhausted. More intensive optimization could entail more drastic changes in the `compute_scores` code (different structuring of data passed around, cythonizing whole classes) and further experimentation with `edge_feature_cache`.

References

[1] Leonard E. Baum and Ted Petrie. "Statistical Inference for Probabilistic Functions of Finite State Markov Chains". In: *The Annals of Mathematical Statistics* 37.6 (1966), pp. 1554–1563. DOI: 10.1214/aoms/1177699147. URL: https://doi.org/10.1214/aoms/1177699147.

[2] Michael Collins. "Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms". In: *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP 2002)*. Association for Computational Linguistics, July 2002. DOI: 10.3115/1118693.1118694. URL: https://aclanthology.org/W02-1001.

[3] Warren Mcculloch and Walter Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 127–147.

[4] David Nadeau and Satoshi Sekine. "A survey of named entity recognition and classification". In: *Lingvisticae Investigationes* 30.1 (2007), pp. 3–26.

[5] Albert B Novikoff. *On convergence proofs for perceptrons*. Tech. rep. STANFORD RESEARCH INST MENLO PARK CA, 1963.

[6] F. Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957. URL: https://books.google.es/books?id=P%5C_XGPgAACAAJ.

[7] Maksim Tkachenko and Andrey Simanovsky. "Named entity recognition: Exploring features." In: *KONVENS*. Vol. 292. 2012, pp. 118–127.