# Natural Language Processing: Problem Set 4

In this programming assignment, you will train a global linear model for named-entity recognition using the perceptron algorithm. In the decoding problem for an input instance $x \in \mathcal{X}$, a global linear model (GLM) uses the following three components: (1) a function that generates possible output structures $\text{GEN}(x)$, e.g. $x$ is a sentence $w_1, \ldots, w_n$ and $y \in \text{GEN}(x)$ is a possible tagging $t_1, \ldots, t_n$, (2) a feature function $f(x, y)$, and (3) a weight vector $v \in R^d$. The decoding problem is defined as

$$y^* = \arg \max_{y \in \text{GEN}(x)} v \cdot f(x, y)$$

We saw in lecture that if the feature function $f$ factors into local features then decoding is tractable even when $\text{GEN}(x)$ is large. For tagging, we partition a tagging into a sequence of histories, $h_1, \ldots, h_n$ and decompose $f$ into a sum of local feature functions over each history and output pair, i.e. $\sum_{i=1}^{n} g(h_i, t_i)$. Our focus will be on trigram tagging, so we define a history as

$$h = \langle t_{-2}, t_{-1}, x, i \rangle$$

The full decoding problem can be described in terms of local features over history/tag pairs as

$$y^* = v \cdot \arg \max_{y \in \text{GEN}(x)} f(x, y) = \arg \max_{t_1, \ldots, t_n \in \text{GEN}(x)} \sum_{i=1}^{n} v \cdot g(\langle t_{i-2}, t_{i-1}, i, x \rangle, t)$$

In this assignment you will experiment with different feature functions and implement the perceptron training algorithm for learning the vector $v$.

## Data

The data and evaluation for this assignment is identical to PA 1.

Recall that we provide a labeled training data set `gene.train`, a labeled and unlabeled version of the development set, `gene.key` and `gene.dev`, and an unlabeled test set `gene.test`. The labeled files take the format of one word per line with each word and tag separated by a space, and and each sentence separated by a blank line.The unlabeled files contain only the words of each sentence and will be used to evaluate the performance of your model.

The task consists of identifying gene names within a biological text. In this dataset there is one type of entity: gene (GENE). The set of tags for this assignment is $\mathcal{T} = \{\text{O}, \text{I-GENE}\}$. The dataset is adapted from the BioCreAtIvE II shared task (`http://biocreative.sourceforge.net/biocreative_2.html`).

## Histories

For the first part of the assignment, we assume the weight vector $v$ is given and fixed. The goal is to solve the GLM decoding problem with this vector. Consider the task of tagging a typical sentence from PA 1

`Characteristics`

```
of
lipase
activity
.
```

The first crucial step is to understand the set of possible histories for the sentence. For instance, for $i = 3$ the possible histories and tag pairs consist of

- $(\langle \text{I-GENE}, \text{I-GENE}, x, 3\rangle, \text{I-GENE})$
- $(\langle \text{O}, \text{I-GENE}, x, 3\rangle, \text{I-GENE})$
- $(\langle \text{I-GENE}, \text{O}, x, 3\rangle, \text{I-GENE})$
- $(\langle \text{O}, \text{O}, x, 3\rangle, \text{I-GENE})$
- $(\langle \text{I-GENE}, \text{I-GENE}, x, 3\rangle, \text{O})$
- $(\langle \text{O}, \text{I-GENE}, x, 3\rangle, \text{O})$
- $(\langle \text{I-GENE}, \text{O}, x, 3\rangle, \text{O})$
- $(\langle \text{O}, \text{O}, x, 3\rangle, \text{O})$

Each of these history tag pairs will map to a different set of local features and a different score.

## Feature Vectors

Once we have a history we can compute its local features. Much of the art of global linear models is selecting good features to describe the data. Feature functions typically look like

$$g_1(\langle t_{-2}, t_{-1}, x, i\rangle, t) = \begin{cases} 1 & \text{if } t_{-2} = \text{O}, t_{-1} = \text{O}, t = \text{I-GENE} \\ 0 & \text{otherwise} \end{cases}$$

which is a local binary feature indicating that the previous two tags are O and the current tag is I-GENE. A common practice is to give features more informative names. For instance, let us call this feature "TRIGRAM:O:O:I-GENE". We will also include features "TRIGRAM:$s$:$u$:$v$" for all $s, u, v \in \mathcal{T}$, so this naming helps organize the feature vector.

Once we have the set of features, we need to be able to compute the feature vector $g(h, t)$ for a history/tag pair $h$, $t$ as well as the inner product $v \cdot g(h, t)$.

- We recommend implementing the feature vector (the output of $g(h, t)$) as a map from strings to counts. Since most features are 0 for a given history, i.e. the local feature vector is sparse, this is more efficient than representing the full vector. For instance if the history/tag pair is $(h, t) = (\langle \text{O}, \text{O}, x, 3\rangle, \text{I-GENE})$, then $g(h, t)$ would contain the mapping "TRIGRAM:O:O:I-GENE" $\rightarrow 1$.

  For this assignment the weight vector $v$ can also be implemented as a map. For instance, in Python the weight vector $v$ might be a dictionary mapping features to weights, e.g.

  ```
  print v["TRIGRAM:O:O:I-GENE"]
  ```

The crucial inner product $v \cdot g(h, t)$ in the decoding problem can then be implemented as a product between these two maps. In Python, this might look like

```
sum((v[k] * val for k, val in g.iteritems()))
```

For more details about this sparse representation, see the notes on log-linear models.

## Decoding

So far we have only discussed local histories and features. To find the best global tagging for a sentence, we solve the following decoding problem:

$$y^* = \arg \max_{y \in \text{GEN}(x)} v \cdot f(x, y) = \arg \max_{t_1, \ldots, t_n \in \text{GEN}(x)} \sum_{i=1}^{n} v \cdot g(\langle t_{i-2}, t_{i-1}, i, x \rangle, t)$$

Luckily this global maximization can be solved using a variation of the Viterbi algorithm. We assume $\mathcal{S}$ is defined the same as with HMMs. The main difference in the algorithm is the scoring function used

**procedure** VITERBIGLM($v$ , $g$, $x$) ▷ $v$ is the weight vector, $g$ is the feature function, $x$ is the sentence
  $\pi(0, *, *) = 0$
  **for** $k = 1 \ldots n$ **do**
    **for** $u \in \mathcal{S}_{k-1}, s \in \mathcal{S}_k$ **do**
      $\pi(k, u, s) = \max_{t \in \mathcal{S}_{k-2}} \pi(k-1, t, u) + v \cdot g(\langle t, u, x, k \rangle, s)$
      $bp(k, u, s) = \arg \max_{t \in \mathcal{S}_{k-2}} \pi(k-1, t, u) + v \cdot g(\langle t, u, x, k \rangle, s)$
  $(t_{n-1}, t_n) = \arg\max_{u \in \mathcal{S}_{n-1}, s \in \mathcal{S}_n} bp(n, u, s) + v \cdot g(\langle u, s, x, n+1 \rangle, \text{STOP})$
  **for** $k = (n-2) \ldots 1$ **do**
    $t_k = bp(k+2, t_{k+1}, t_{k+2})$
  **return** $t_1, \ldots t_n$

Note that the GLM Viterbi algorithm requires computing the local score $v \cdot g(\langle t, u, x, k \rangle, s)$. We recommend abstracting this part out of the function and computing it as described in the previous section. The rest of the code is very similar to HMM Viterbi. You should be able to repurpose the code from PA 1 for this part of the assignment.

## Part 1 (1 points)

In this problem you will experiment with building a trigram tagging decoder with a pre-trained model.

- Consider a very simple model with two types of feature functions for all tags $s$, $u$, $v$ and words $r$

$$g_{\text{TRIGRAM:s:u:v}}(\langle t_{-2}, t_{-1}, x, i \rangle, t) = \begin{cases} 1 & \text{if } t_{-2} = s \text{ and } t_{-1} = u \text{ and } t = v \\ 0 & \text{otherwise} \end{cases}$$

$$g_{\text{TAG:u:r}}(\langle t_{-2}, t_{-1}, (w_1 \ldots w_n), i \rangle, t) = \begin{cases} 1 & \text{if } w_i = r \text{ and } t = u \\ 0 & \text{otherwise} \end{cases}$$

Consider the example sentence ("Characteristics of ...") as $x$, we extract the following feature vector (as a map from features to counts) from an example history/tag pair

$$g(\langle \text{O}, \text{O}, x, 3 \rangle, \text{I} - \text{GENE}) = \{\text{"TRIGRAM:O:O:I-GENE"} \rightarrow 1, \text{"TAG:I-GENE:lipase"} \rightarrow 1\}$$

For this problem we provide a pre-trained weight vector $v$ for these features (initialized $v = 0$ and trained for $k = 5$ iterations). The file `tag.model` contains the vector. All lines in the file consist of two columns `FEATURE`, `WEIGHT` indicating $v(\text{FEATURE}) = \text{WEIGHT}$. More specifically.

- Lines of the form `TAG:lipase:I-GENE 0.23` mean the feature `TAG:lipase:I-GENE` representing lipase tagged with I-GENE has weight $0.23$.
- Lines of the form `TRIGRAM:O:O:I-GENE 0.45` mean the feature `TRIGRAM:O:O:I-GENE` representing the trigram O, O, I-GENE has weight $0.45$.

- The goal of this problem is to decode with this model. First read `tag.model` into a map from feature strings to weights. Next for each sentence in development data run the Viterbi algorithm as described above.

  Write your output to a file called `gene_dev.p1.out` and locally evaluate by running

  ```
  python eval_gene_tagger.py gene.key gene_dev.p1.out
  ```

  When you are ready to submit, run your model on `gene.test` and write the output to `gene_test.p1.out`. Run `python submit.py` to submit.

## Perceptron Training

The next two parts of the assigment focus on estimating the weight vector $v$ using the perceptron algorithm. Note that unlike HMMs we cannot simply read off the parameters from the training data. We will instead repeatedly solve the decoding problem on the training data to estimate better parameters.

Training the tagger requires a corpus of tagged examples $(x^{(i)}, y^{(i)})$ for $i \in \{1, \ldots, M\}$. The file `gene.train` contains these training sentences.

Consider the tagged version of the training data above.

```
Characteristics O
of O
lipase I-GENE
activity O
. O
```

We can see that $(\langle \text{O}, \text{O}, x, 3 \rangle, \text{I-GENE})$ is the correct history/tag pair for position $i = 3$.

The perceptron algorithm is defined in detail in the Week 10 class slides. The algorithm looks a bit complicated, but once you understand it, the code should be very simple.

We start with $v = 0$, then for each sentence/tag pair $(x, y)$ in the training data in order we run the following update

1. Compute the best tagging, $z = \arg\max_{t_1,\ldots,t_n \in \text{GEN}(x)} \sum_{i=1}^{n} v \cdot g(\langle t_{i-2}, t_{i-1}, i, x\rangle, t)$
2. Compute the best tagging feature vector, $f(x, z)$
3. Compute the gold tagging feature vector, $f(x, y)$
4. Update the weights, $v \leftarrow v + f(x, y) - f(x, z)$

The first step should make use of Part 1. The second and third steps should use your helper function from above. The final step is vector arithmetic.

## Part 2 (1 point)

A major reason that the model from Part 1 performs so poorly is that it does nothing to handle rare words. We could introduce _RARE_ tokens or word classes; however, a major benefit of GLMs is that we can instead introduce features that target specific properties of the words.

A good example of this type of feature is word suffixes. We saw in PA1 that suffixes can be a useful indicator as a word class, so we expect they will be informative features. The new features are, for all suffixes $u$, tag $v$, and lengths $j \in \{1, 2, 3\}$

$$g_{\text{SUFF:u:j:v}}(\langle t_{-2}, t_{-1}, (w_1 \ldots w_n), i\rangle, t) = \begin{cases} 1 & \text{if } u = \text{suffix}(w_i, j) \text{ and } t = v \\ 0 & \text{otherwise} \end{cases}$$

where $\text{suffix}(w, j)$ returns the last $j$ letters of $w$.

- For this question, you should first implement the perceptron algorithm to estimate a weight vector $v$ for the new set of features. Start with $v = 0$ and run $K = 5$ iterations. Follow the steps in the Perceptron Training section. When your code is finished write the final model out to `suffix_tagger.model`.
- After training is complete, run your decoder from Part 1 (with the suffix features) and then run on the development set.

  When you are ready to submit, run your model on `gene.test` and write the output to `gene_test.p2.out`. Run `python submit.py` to submit.

## Part 3 (1 Point)

As mentioned above, the benefit of this formulation is the ability to add arbitrary features conditioned on the input sentence $x$. In the last question we experimented with adding suffix features from $i$-th word of the sentence. Many other features have been shown to be useful for tagging.

- For this question you should run tagging experiments with custom features. There are many ways to come up with new features. One strategy is to look at the errors your tagger made in Part 2 and try out features that target specific issues. We also encourage you to look up features in the NLP literature or in open-source taggers. Don't be discouraged if adding seemingly useful features sometimes decreases the accuracy of your tagger; feature selection can be a difficult problem.

  When you are ready to submit, run your model on `gene.test` and write the output to `gene_test.p3.out`. Run `python submit.py` to submit.