CSE250B Project 2

Mehmet Koc A53035914
Zeynep Su Kurultay A53034989

February 16, 2014

Abstract

In this project, we complete a task of punctuation prediction via linear-chain Conditional Random Fields (CRF) which is a special case of log-linear model. In the task, each word in the dataset is associated with one punctuation tag. Thus, the task is to predict the most likely punctuation tag each time. The dataset is obtained from Enron Email Dataset [1]. Two optimization algorithms are used for the CRF model, namely Collins Perceptron and Stochastic Gradient Ascent. The accuracies obtained for CRF with Collins Perceptron and for CRF with Stochastic Gradient Ascent are 93.40 and 92.69, respectively.

1 Introduction

In this project, a punctuation model needs to be fit for the words of a document using a training set. Best model parameters are selected by sparing a validation set in the training set. Then, the performance of the constructed model is measured on a separate test set.

Each word in the given datasets is associated with one puncutation tag, so the model needs to predict a tag (one from the set ["COMMA", "PERIOD", "QUESTION MARK", "EXCLAMATION MARK", "COLON", "SPACE"]) for each word. A suitable model for this purpose is linear-chain CRF. This model utilizes an underlying log-linear model as follows:

$$p(y|x;w) = \frac{exp(w^T F(x,y))}{\sum_{\forall y} exp(w^T F(x,y))} = \frac{exp(w^T F(x,y))}{Z(x,w)}$$

where x is the $n \times 1$ observation vector (sentence in our case with length n), y is the $n \times 1$ tag vector for the words in x, $F(x,y)$ is $J \times 1$ feature function (FF) vector and w is the $J \times 1$ weight vector for FFs.

Note that the denominator in the expression above is a sum over all possible y vectors. This sum will be important to normalize the probabilities and it is challenging without an assumption on the form of FFs and proper algorithm since it is $O(m^n)$ in the general case where m is the number of possible tags. This issue is addressed in Section 2.1. The learning procedure for learning w is highlighted in Section 2.2 and 2.4 and the inference procedure in Section 2.3.

Each FF consisting of the sum over x using its low-level FF f_j at index i is associated with the entire observation x, and two adjacent labels y_i and y_{i-1}. As dictated in the project specifications, we prepare FF templates in the following fashion:

$$F_j(x, y) = A_a(x)B_b(y)$$

In this equation, A part ranges over functions of input while B part ranges over functions of candidate label values. In this project F_j are selected such that each of the parts of FFs are indicator functions. This means each low-level FF f_j is either 0 or 1 and one does not have to normalize features to obtain the optimal performance. Taking into account the general definition of FFs:

$$F_j(x, y) = \sum_{i=1}^{n} f_j(y_{i-1}, y_i, x, i)$$

We need to make some alterations on our original function, as so:

$$f_j(y_{i-1}, y_i, x, i) = A_a(x, i)B_{b,c}(y_{i-1}, y_i, i)$$

One advantage to using CRF is that the usage of arbitrary overlapping features can be exploited. This allows the trainer to define FFs that represent different punctuation rules which have many overlapping qualities. FFs are described in detail in the Experiments section.

2 Algorithm

2.1 Efficient Algorithm for Normalizing Constant

As mentioned in the Introduction section, we need an efficient procedure to compute $Z(x, w)$ so that we can normalize probabilities and use it to measure the performance of different parameters of our model. Let's start by defining the forward vector:

$$\alpha(k, v) = \sum_{y_1} \sum_{y_2} ... \sum_{y_{k-1}} exp(\sum_{i=1}^{k-1} g_i + g_k(y_{k-1}, v))$$

$$\alpha(k, v) = \sum_{y_{k-1}} \sum_{y_1} \sum_{y_2} ... \sum_{y_{k-2}} exp(\sum_{i=1}^{k-2} g_{k-2} + g_{k-1}(y_{k-2}, y_{k-1}))exp(g_k(y_{k-1}, v))$$

$$\alpha(k, v) = \sum_{u} \alpha(k-1, u)exp(g_k(u, v))\, where\, \alpha(1, v) = exp(g_1(START, v))$$

One can start recursing from $\alpha(1, v)$ to obtain $\alpha(n, v)$ and in the end:

$$Z(x, w) = \sum_{v} \alpha(n, v)$$

where the effect of STOP tag is assumed to be included in $g_n(y_{n-1}, y_n)$.

Since the recursive formula for $\alpha(k, v)$ multiplies and sums a lot of exponentials, it is better to keep to computations in $log\alpha(k, v)$ and in the $logZ(x, w)$ to avoid overflow/underflow. Then, using log-sum-exp trick [2]:

$$log\alpha(k, v) = log(\sum_{u} exp(log\alpha(k-1, u) + g_k(u, v)))$$

Let $log\alpha(k-1, u) + g_k(u, v)$ be $h_1(k, u, v)$.

$$log\alpha(k, v) = \max_{u} h_1(k, u, v) + log(\sum_{u} exp(h_1(k, u, v) - \max_{u} h_1(k, u, v)))$$

Similarly, for $Z(x, w)$:

$$logZ(x, w) = log(\sum_{v} exp(log\alpha(n, v)))$$

Let $exp(log\alpha(n, v))$ be $h_2(v)$.

$$logZ(x, w) = \max_{v} h_2(v) + log(\sum_{v} exp(h_2(v) - \max_{v} h_2(v)))$$

The procedure highlighted in this section takes $O(mn)$ time if one assumes $g_i \, \forall i$ has been calculated before and the complexity of exp and log operations are assumed to be $O(1)$.

2.2 Learning in CRF with Collins Perceptron

Having a training set of $\{x_i, y_i\}_{t=1}^{t}$ of T sentences and their tags, the objective is to maximize:

$$LCL_{training} = \sum_{t=1}^{T} logp(y_t|x_t; w) = \sum_{t=1}^{T} w^T F(x_t, y_t) - logZ(x_t, w)$$

where x_t and y_t are the word vector and the corresponding tag vector for t^{th} training example. So let's calculate:

$$\frac{\partial logp(y_t|x_t; w)}{\partial w} = F(x_t, y_t) - E_{p(y'|x;w)} \{F(x_t, y')\}$$

The derivative above cannot be equated to 0 and solved in closed form. Instead, one can use an iterative stochastic gradient method to find the optimal w. In this scenario, there is still one more problem evaluating the expectation $\forall y$. One solution for this is using the Collins perceptron method as follows where p is the iteration number which uses t^{th} training example.

$$y_p* = \operatorname*{argmax}_{y_t} logp(y_t|x_t; w_t)$$

Inferring y_p* is discussed in Section 2.3. Collins perceptron puts all probability mass on y_p* and does the update where update also includes a regularization term:

$$w_{p+1} \leftarrow w_p + \lambda(F(x_t, y_t) - F(x_t, y_p*) - 2\mu w_p)$$

After finding w^* for (λ, μ) pair, its performance can be tested on a validation set $\{x_r, y_r\}_{r=1}^{R}$ and (λ, μ) that maximizes $LCL_{validation}$ can be chosen:

$$LCL_{validation} = \sum_{r=1}^{R} logp(y_r|x_r; w) = \sum_{r=1}^{R} w^T F(x_r, y_r) - logZ(x_r, w) \, where \, w = w^*$$

The time complexity of learning with T examples is $O(m^2JnP)$ for each validation parameter pair if one asssumes computing $F(x, y)$ takes $O(1)$ time. Note that in this expression, P is the number of iterations performed where $P = Te$ where e is epoch number. This $O(P) = O(Te)$ can be reduced if early stopping is applied. Also note that $O(m^2Jn)$ in $O(m^2JnP)$ comes from the inference which is discussed in Section 2.3.

2.3 Inference in CRF

Assuming that one has the parameter w for CRF model, inference requires finding the best sequence y* such that

$$y* = \underset{y}{\operatorname{argmax}} \, p(y|x; w) = \underset{y}{\operatorname{argmax}} \, w^T F(x, y)$$

In the most general case, the problem above is exponential. However, with a special form of $F_j(x, y)$, the complexity can be reduced greatly by dynamic programming.

$$F_j = \sum_{i=1}^{n} f_j(y_i, y_{i-1}, x, i) \, where \, y_0 = START \, and \, y_{n+1} = STOP$$

$$y* = \underset{y}{\operatorname{argmax}} \sum_{j=1}^{J} w_j F_j(x, y) = \sum_{i=1}^{n} \sum_{j=1}^{J} w_j f_j(y_i, y_{i-1} x, i) = \sum_{i=1}^{n} g_i(y_i, y_{i-1})$$

If we define U(k,v) as follows, then by dynamic programming:

$$U(k, v) = \max_{y_1, \cdots y_{k-1}} \sum_{i=1}^{k-1} g_i(y_i, y_{i-1}) + g_k(y_{k-1}, v)$$

$$U(k+1, v) = \underset{U}{\operatorname{argmax}} U(k, v) + g_{k+1}(u, v) \, where \, U(1, v) = g_1(START, v)$$

Starting with the base case, one can obtain U(n,v) $\forall v$ in $O(m^2Jn)$ time. Then,

$$y_n* = \underset{v}{\operatorname{argmax}} \, U(n, v)$$

This only provides the last label of the best sequence. To obtain the rest, one can create Q(k,v) in the forward step while calculating U(k,v) and then backtrack in the end from y_n*.

$$Q(k, v) = \underset{u}{\operatorname{argmax}} \, U(k+1, v) \, for \, k = 1 \, to \, n-1$$

$$y_{k-1}* = Q(k-1, y_k*)$$

Assuming Q(k,v) is constructed when constructing U(k,v), the backtracking is just n lookups and takes O(n) time. Hence, the total time complexity for inference is $O(m^2Jn)$.

2.4 Learning in CRF with Stochastic Gradient Ascent (SGA)

Remember from Section 2.2 that the gradient of LCL is:

$$\frac{\partial log p(y_t|x_t;w)}{\partial w} = F(x_t, y_t) - E_{p(y'|x;w)}\{F(x_t, y')\}$$

Instead of putting all the probability mass on the most likely tag vector y_p^* for the current example x_t and then doing the stochastic gradient update, one can also try approximating the expectation using a sampling method. One possible method for this purpose is Gibbs sampling [3].

In Gibbs sampling, an arbitrary initial guess is selected for an entire label y, like so: $\{y_1, ...y_n\}$. Then, for each y value, a new value is drawn, as shown in the following conditional distribution: $y_1 \leftarrow p(Y_1|x, y_2, ..., y_n; w)$, $y_2 \leftarrow p(Y_2|x, y_1, y_3, ..., y_n; w)$ etc. In theory, when this drawing phase for each value of y is repeated infinitely, the distribution of $y = \{y_1, ..., y_n\}$ converges to the true distribution $p(y|x; w)$. Since this is not possible in practice, this step is repeated a finite number of times, and the number of repetitions is decreased each time. These two mentioned practices allow the final value to be less dependent on the starting point and the samples to be independent of each other, respectively.

Doing one round of Gibbs sampling with the efficient algorithm in [3] is $O(mn)$. After S samples are selected in $O(mnS)$ time, then one can approximate the expectation using those examples are doing the same parameter update as in Section 2.2.

3 Experiments

3.1 Preprocessing on Data

The training set that we use is taken from The EnronSent Corpus [1]. There are 70115 sentences in the training set and 28027 sentences in the test set. We do some minor transformations on the data provided in order to make processing easier. First, we code the labels in the training and test data. Since the cardinality of possible labels is 6, we code each of the possible tags from 1 to 6. This way, things are more convenient mathematically and the process of obtaining accuracy is easier, since instead of string comparison we are able to compare two integers. Second, we randomly shuffle the given datasets in order to get rid of any biased order that might affect the training process.

Instead of using the datasets in its unprocessed form to generate features, we use Stanford's POS tagger [4]. Thus, we process the datasets so that words like "don't" can be handled accordingly in the POS tagger that we use. This POS tagger does not handle such words well, outputting "NN" (noun) for "dont" and "VBP RB" (verb particle) for "don't". Even though the second output is correct, it causes an imbalance in the given labels for the set. So, we go through the dataset to separate such instances in advance and add the appropriate label.

We choose to use one of the trained models provided in the POS tagger, namely wsj-0-18-bidirectional-distsim.tagger, which uses bidirectional architecture and

Type	Training Set	Test Set
comma	28555	10666
period	57260	22493
question mark	10904	4792
exclamation point	2567	1066
colon	1009	1174
space	580082	235549

Table 1: Counts of each of the labels in the training and test sets.

includes word shape and distributional similarity features. This model is reported to have best accuracy among the available trained POS tagger models (97.28% correct within the domain of the training set and 90.46% correct on unknown words), so we choose this one.

After transforming both the training and the test sets as explained so far, we separate the whole training set randomly into two sets, namely training and validation, with a ratio of 7:3.

Some characteristics of the data can be seen in Table 1. This Table shows the counts of each of the labels in the training and test sets (The counts in the test set are only provided for information). Note that the percantages of the tags other than SPACE are very low in the training set. This indicates using 0/1 accuracy to evaluate the results is not a good idea. That is why confusion matrices are included in Results section.

3.2 Feature Function Generation

We implement the FF generation with the methodology explained in the Introduction section. In generating FFs, we employ a mixture of English punctuation rules implied from the results of POS tagger and automatically generated FFs obtained via templates. We define 70 rules that represent the common practice of punctuation and also have 36 FFs that lets the model assign weights to each successive punctuation label. Note that among the 70 FFs, there are FFs that address the unlikely scenarios of punctuation to improve recall metric for each tag. Normally, a combined unigram and bigram model could report the best accuracy for this particular task, but this approach would result in generating too many FFs. This would cause the computational complexity to increase drastically. so we yield to the tradeoff and use this approach.

Using the flexibility of the definition of FFs, we employ different kinds of FFs to suit different rules' needs. The A part of the FF has a variety of combinations, like $A_a(x,i) = I(POS(i) = a)$, $A_{a,b}(x,i) = I(POS(i) = a)\, I(POS(i-1) = b)$ and the B part likewise can look at the current label and the previous label. Since the arguments contain the whole sentence and current position, we also have the chance to represent rules concerning arbitrary positions in the sentence, say "If the first word is a member of a question group (why, how etc)?" to predict the last label as QUESTION MARK.

3.3 Collins Perceptron

The Collins perceptron method described in Algorithm section requires two parameters, namely λ and μ. The grid search for λ and μ is conducted over the values:

$$\mu = [5^{-2}, 5^{-1}, 5^{0}, 5^{1}, 5^{2}] \quad \text{and} \quad \lambda = [10^{-2}, 10^{-1}, 10^{0}, 10^{1}]$$

After we train on the training set, we find the best values on the validation set (the metric is $LCL_{validation}$) and use these values to obtain the results on the test set. Note that the same grid search values are also used for SGA.

3.4 SGA

For the second part of the project, we use SGA with Gibbs Sampling ($S_{max} = 1000$) in order to optimize the objective function. We use checkgrad2.m [5] to make sure that the partial derivative is approximated well. Our implementation of SGA strives to maximize word level accuracy. For the computational strain to be somewhat reduced, we implement early stopping, which means checking for the increase in accuracy at regular intervals and stopping trying to optimize it if the difference between two checks is low enough. The optimal values for λ and μ for the update rule specified in Section 2.2 are grid-searched and the best values obtained on the validation set are applied to the test set, as mentioned. The values obtained are reported in the Results section.

4 Results

After grid searching the optimal hyperparameters for Collins Perceptron, the best values are found to be $\lambda = 10^{-2}$ and $\mu = 5^{-1}$. Note that the best learning rate is on the edge of grid search values for λ. This is normal since as the learning rate drops, stochastis gradient methods perform better. The drawback is that training takes longer with a lower learning rate. Almost the same hyperparameter values are also optimal for SGA ($\lambda = 10^{-2}$ and $\mu = 5^{0}$) as expected since both methods exploit gradient in a stochastic fashion using the same training examples. The only difference between them how they approximate the expectation and this difference decay in each iteration as w converges to the best value. The test 0/1 accuracy results per labels for CRF trained with SGA and CRF trained with Collins Perceptron can be seen in Table 2 and 3. Overall accuracies for each model are 92.69% and 93.40%, respectively.

If one came up with a classifier that predicts SPACE for each word but the last words and predicts PERIOD for the last words, the 0/1 accuracy would be about 93% (baseline). Although this accuracy looks about the same for our accuracy, our classifier is superiour in the sense that it can also predict the tags for punctuation marks other than SPACE and PERIOD.

As can be seen from the tables, the accuracies for predicting colon and exclamation point are very low. This can be attributed to their scarcity in the given dataset, so the models cannot be trained well enough. Moreover, when we examined some instances of examples containing these labels, we see that

Type	Test Set Accuracy Percent
comma	39.74
period	97.84
question mark	38.53
exclamation point	0.00
colon	0.2
space	96.59

Table 2: Test set accuracy of CRF trained with SGA

Type	Test Set Accuracy Percent
comma	40.35
period	98.12
question mark	35.64
exclamation point	0.00
colon	0.1
space	97.41

Table 3: Test set accuracy of CRF trained with Collins Perceptron

the sentences mostly do not adhere to the general rules of English (which we used to define our FFs). For example, most of the sentences that end with an exclamation point do not have interjections in them (like "Oh!"). This might be another reason that the accuracy is low for these labels, that the sentences in the dataset are not representative of the general rules of English that were used to define the FFs used to train the models.

One other assumption that is implicitly made when training the models is that each line of the inputs contain one example. However, this is not the case due to some noise in the data. For example, the line "Lucy says I sent you the roll did you get it" contains two examples and will not adhere to the FF which looks for a verb-role at the beginning of a sentence in order to classify a question mark.

5 Conclusion

In conclusion, we train two linear-chain CRF models, optimized by the Collins perceptron and SGA algorithms on a given sentence and label dataset in order to predict the appropriate punctuation labels. No normalization is needed since all low-level FFs are binary. To generate features, we get help from the POS tagger in [4] since coming up with exhaustive set of features that consider many rules of English texts is very hard.

In the learning phase, we do grid search for hyperparameters and apply the best values we get from validation to the test set. Furthermore, L_2-regularization is applied to prevent overfitting. The accuracies of the trained models with Collins Perceptron and SGA algorithms are 93.40% and 92.69%, respectively. These accuracies obtained are close to the baseline. However, classifying the baseline tagging of every word as either SPACE or PERIOD is not meaningful since the classifier needs to predict tags for a greater set of punctuation marks.

Our classifiers are able to classify tags other than SPACE and PERIOD. The low accuracy in our classifier can be attributed to some noise in the data, sparseness of some labels and computational limits preventing the model to train on more FFs.

References

[1] http://verbs.colorado.edu/enronsent/

[2] https://hips.seas.harvard.edu/blog/2013/01/09/computing-log-sum-exp/

[3] http://cseweb.ucsd.edu/ elkan/250B/CRFs.pdf

[4] http://nlp.stanford.edu/downloads/tagger.shtml

[5] http://people.csail.mit.edu/jrennie/matlab/checkgrad2.m