HW2: Tagging From Scratch

Jeffrey Ling
jling@college.harvard.edu

Rohil Prasad
prasad01@college.harvard.edu

February 22, 2016

1 Introduction

In this assignment, we tackle the problem of part-of-speech tagging.

The problem can be outlined as follows. Given a set of tokenized text with each token given a tag corresponding to the part of speech that it functions as, can we train a model that can read in other sets of tokenized text and accurately determine the part of speech for every word?

We will train our models on a tagged excerpt from the Wall Street Journal and test them on another, i.e. the Penn Treebank dataset. We examine three different models for this purpose. These three models are multinomial naive Bayes, logistic regression, and the neural network from Collobert et al (2011).

In Section 2, we introduce our notation and give a description of our problem in formal language. In Section 3, we describe our three models. In Section 4, we present our experimental results for model performance and hyperparamater tuning. In Section 5, we conclude with a discussion of our results.

2 Problem Description

Our raw data consists of pairs $(w_i, t_i) \in \mathcal{V} \times \mathcal{T}$ for $i = 1, \dots, N$ where N is our data size, where $w_i \in \mathcal{V}$ is a word in our corpus and t_i is its corresponding part of speech (POS) tag. The t_i are elements of a finite set of classes $\mathcal{T}$, which we take as the list of POS tags provided in the homework. Example POS tags are NNP, VBD, VBZ, etc. and in this problem we have a total of $|\mathcal{T}| = 45$ tags.

We construct a feature representation of each word w as follows.

First, construct the vocabulary $\mathcal{V}$ of the training and testing sets. To keep this from growing too large, we only use lower-case words. Furthermore, we map any uncommon words to a special "RARE" word, where we use the words in the publicly available data from the Glove word embeddings as our "common words" dictionary. A word w then can be associated with a sparse one-hot feature vector $\boldsymbol{j} \in \{0,1\}^{|\mathcal{V}|}$, with 1 at the index j that w maps to in $\mathcal{V}$ (where $j = 1, \dots, |\mathcal{V}|$).

Second, we can also construct a feature $c \in \mathcal{C}$ that keeps track of the capitalization information in w that we lost above when we made it lowercase. This feature is dependent on if w is all lowercase, all uppercase, starts with an uppercase letter, or contains an uppercase letter, in that order of priority, thus giving $|\mathcal{C}| = 4$. We represent this feature for w with a sparse one-hot feature vector $\boldsymbol{c} \in \{0,1\}^{\mathcal{C}}$.

Finally, we add context by not only adding in the features of w_i, but also adding in the features of the words in a *window* around w_i. In our case, we use a fixed window of size 5, so our data sample becomes $\boldsymbol{x}_{i,word} = \{\boldsymbol{j}_{i-2}, \boldsymbol{j}_{i-1}, \boldsymbol{j}_i, \boldsymbol{j}_{i+1}, \boldsymbol{j}_{i+2}\} \in \{0,1\}^{5|\mathcal{V}|}$ and $\boldsymbol{x}_{i,cap} = \{\boldsymbol{c}_{i-2}, \boldsymbol{c}_{i-1}, \boldsymbol{c}_i, \boldsymbol{c}_{i+1}, \boldsymbol{c}_{i+2}\} \in \{0,1\}^{5|\mathcal{C}|}$ of vocabulary and capitalization features corresponding to the surrounding word window, and we will denote these features $(\boldsymbol{x}_{i,word}, \boldsymbol{x}_{i,cap})$ by $\boldsymbol{w}_i$.

If the window overlaps the start or end of the sentence, we represent the words as a vector of all zeroes.

Our goal is thus to learn a function $f : \{0,1\}^{5|\mathcal{V}|} \times \{0,1\}^{5|\mathcal{C}|} \to \mathbb{R}^{|}\mathcal{T}|$ that takes in a window of words $\boldsymbol{w}$ and outputs a prediction vector $\hat{\boldsymbol{y}}$ where $\hat{y}_k$ is the log probability $\log p(t|w_i)$ of the middle word w_i of $\boldsymbol{w}_i$ having tag i.

2.1 Practice

In practice, we simply represent a word $w \in \mathcal{V}$ with its index j where $j = 1, \dots |\mathcal{V}|$, and similarly represent its feature $c \in \mathcal{C}$ with the index $j_c = 1, \dots, |\mathcal{C}|$. The `nn.LookupTable` function in Torch allows us to perform the matrix operation $\boldsymbol{j}W$ by simply sparsely indexing into the jth row of W, and we do the same for the capitalization features.

2.2 Extension: Suffix Features

As an extension to the baseline model described in Collobert et al, we add in suffix features as well. We merely take for a word w its last two letters $s \in \mathcal{S}$ where $\mathcal{S}$ is the set of word suffixes. We can then derive features $\boldsymbol{s} \in \{0,1\}^{\mathcal{S}}$ for word w and $\boldsymbol{x}_{i,suffix} \in \{0,1\}^{5|\mathcal{S}|}$ for a window of words centered at i just as described before, and concatenate them to the feature representation.

Again, in practice we store the index $j_s = 1, \dots, |\mathcal{S}|$ for word w, and use `nn.LookupTable`.

3 Model and Algorithms

We use three models:

- multinomial naive Bayes (MNB)
- logistic regression
- neural network from Collobert et al (2011)

Below we describe the setup of each.

3.1 Multinomial Naive Bayes

In MNB, we assume that the features of a word window $\boldsymbol{w}$ are independent of each other.

We can apply Bayes' law to calculate the probability $p(t|\boldsymbol{w})$ of class t given $\boldsymbol{w}$.

$$p(t|\boldsymbol{w}) = \frac{p(t)p(\boldsymbol{w}|t)}{p(\boldsymbol{w})}$$

Our prediction $\hat{y}$ for $\boldsymbol{w}$ is given by the maximum-probability tag

$$\hat{y} = \arg\max_t p(t|x)$$

Note that $\boldsymbol{w}$ is a window of words. Assuming that the middle word is equal to w_i, we can apply our independence assumption to get

$$p(t|\boldsymbol{w}) = \frac{p(t)\prod_{d=-2}^{d=2} p(w_{i+d}|t)}{\prod_{d=-2}^{d=2} p(w_{i+d})}$$

The $p(w_{i+d})$ are constant functions of the output t, so we can focus on calculating the terms in the numerator.

Recall that every word w can be represented as a two sets of binary values $e_1, \ldots, e_{|\mathcal{V}|}$ and $e_1^{cap}, \ldots, e_{|c|}^{cap}$. By our independence assumption, we have

$$p(w|t) = p(e_1, \ldots, e_{|\mathcal{V}|}, e_1^{cap}, \ldots, e_{|\mathcal{C}|}^{cap}|t) = \prod_{m=1}^{|\mathcal{V}|} p(e_m|t) \prod_{n=1}^{|\mathcal{C}|} p(e_n^{cap}|t)$$

Therefore, it suffices to maximize the $p(e_m|t)$ and $p(e_n^{cap}|t)$. Under negative log-likelihood loss, these are maximized over the entire training set by counting the number of samples with tag t that have word feature e_m and capitalization feature e_n^{cap} respectively and then dividing by the total number of samples with tag t.

The maximum likelihood estimate for $p(t)$ is the proportion of samples with tag t.

Overall, our prediction becomes

$$\hat{y} = \arg\max_t \sum_d (\sum_m \log p(e_m|t) + \sum_n \log p(e_n^{cap}|t)) + \log p(t)$$

where the sum is taken over all words in the window $\boldsymbol{w}$.

3.2 Logistic Regression

We assume a linear model $z = \boldsymbol{x}_{word}\boldsymbol{W} + \boldsymbol{x}_{cap}\boldsymbol{W}_{cap} + \boldsymbol{b}$, where $\boldsymbol{x}_{word}$ and $\boldsymbol{x}_{cap}$ are the word and capitalization features of an input word window $\boldsymbol{w}$.

We then apply the softmax function to get an output $\hat{\boldsymbol{y}}$, where $\hat{y}_t$ is the predicted probability for a tag t given the window $\boldsymbol{w}$.

The parameters are optimized via stochastic gradient descent and a negative log-likelihood criterion like in our last assignment.

3.3 Neural Network

The neural network model of Collobert et al (2011) simply stacks more functions before the softmax in logistic regression. A diagram:

The first layer of the model performs a matrix multiplication for each one-hot $\boldsymbol{j}_i$ as $\boldsymbol{j}_i W_{0,word}$, where $W_{0,word}$ is a $|\mathcal{V}|$ by d_{word} to get a word embedding $\boldsymbol{e}_i \in \mathbb{R}^{d_{word}}$. Here, d_{word} is a hyperparameter that can be chosen.

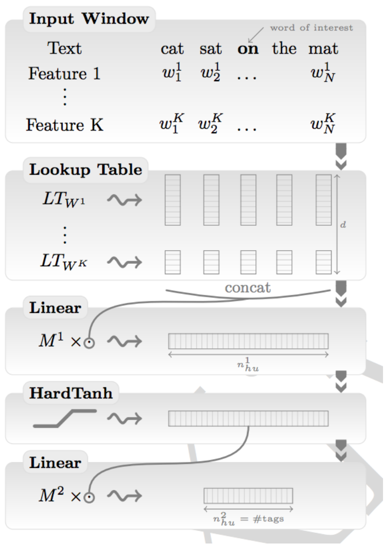

Figure 1: Neural network architecture of Collobert et al.

We do the same with a weight matrix $W_{0,cap}$ of size $|\mathcal{C}|$ by d_{cap} on $\boldsymbol{c}_i$ to get a capitalization vector $\boldsymbol{e}_{i,cap} \in \mathbb{R}^{d_{cap}}$.

Thus the input $(\boldsymbol{x}_{word}, \boldsymbol{x}_{cap})$ is transformed into a pair

$$[\boldsymbol{e}_{i-2}, \boldsymbol{e}_{i-1}, \boldsymbol{e}_i, \boldsymbol{e}_{i+1}, \boldsymbol{e}_{i+2}] \in \mathbb{R}^{5d_{word}}$$

$$[\boldsymbol{e}_{i-2,cap}, \boldsymbol{e}_{i-1,cap}, \boldsymbol{e}_{i,cap}, \boldsymbol{e}_{i+1,cap}, \boldsymbol{e}_{i+2,cap}] \in \mathbb{R}^{5d_{cap}}$$

Let us represent the concatenation as $\boldsymbol{z}_{0,i} \in \mathbb{R}^{5d_{word}+5d_{cap}}$. We then perform a linear transformation with matrix W_1 of size $5d_{word} + 5d_{cap}$ by h, where h is a hyperparameter that's the number of hidden layers, and also use a bias term $b_1 \in \mathbb{R}^h$ to get a result $\boldsymbol{z}_{1,i} \in \mathbb{R}^h$.

$$\boldsymbol{z}_{1,i} = \boldsymbol{z}_{0,i} W_1 + b_1$$

We then perform the hard tanh activation function termwise, which is defined as

$$HardTanh(x) = \begin{cases} -1 & x < -1 \\ x & -1 \leq x \leq 1 \\ 1 & x > 1 \end{cases}$$

Finally, we perform a linear transformation with matrix W_2 of size h by $|\mathcal{T}|$ to get the desired output $\boldsymbol{z}_{2,i} \in \mathbb{R}^{|\mathcal{T}|}$. The last step is a log softmax.

$$\boldsymbol{z}_{2,i} = HardTanh(\boldsymbol{z}_{1,i}) W_2 + b_2$$

$$\widehat{\boldsymbol{y}}_i = LogSoftMax(\boldsymbol{z}_{2,i})$$

We also optimize the parameters with SGD and backpropagation.

3.3.1 Glove Embeddings

Glove word embeddings are pretrained word vectors from Pennington et al (2014). We can initialize our word embedding parameters in the neural network architecture with these word embeddings to obtain better results, since we hope that SGD will reach a better local optimum in the parameter space with previous substructure.

3.4 Training Methods

We ran vanilla batch stochastic gradient descent on training and validation sets. For batches, we shuffled the training set once and iterated over it for each epoch. For preliminary results we ran SGD for at least 5 epochs, and then stopped when relative validation loss (defined as the fractional change in validation loss) dropped below 0.1%.

We continued training on the neural model (with Glove word embeddings) when it was clear that the model had not completed training with this criterion. The use of validation error rather than training error prevents overfitting.

4 Experiments

We ran batch SGD for maximum 20 epochs with batch size 32 and learning rate 0.01. For naive Bayes we used $\alpha = 1$. For neural models, we used word embedding dimension 50, capitalization feature embedding dimension 5, and hidden layer size 300 (as in Collobert et al). We ran most experiments on Odyssey with 16GB memory.

We didn't extensively tune hyperparameters in this experiment, but rather tried different model setups.

Training time tended to vary for each model depending on the Odyssey server it ran on and time of day. Table 1 gives some ballpark timing results for each model.

Model	Timing
Naive Bayes	16 s
Logistic Regression (per epoch)	10 min
Neural (per epoch)	12 min

Table 1: Timing results on PTB training.

For reference, the PTB dataset has 950885 training points, so a batch size of 32 gives a training time of 24.2 ms per batch.

Figure 2 shows the training and validation loss over epochs for the neural model with Glove word embeddings. Note that loss consistently decreases, and we stopped training when validation loss stopped decreasing. The bump in the validation accuracy is from when we restart training from a saved model, and is possibly due to a new batch shuffle for training.

We found that after 20 epochs, the neural model with pretrained word embeddings had the best validation rate, but training had not yet converged. Thus, we ran it for about 10 more epochs until the validation error stopped decreasing.

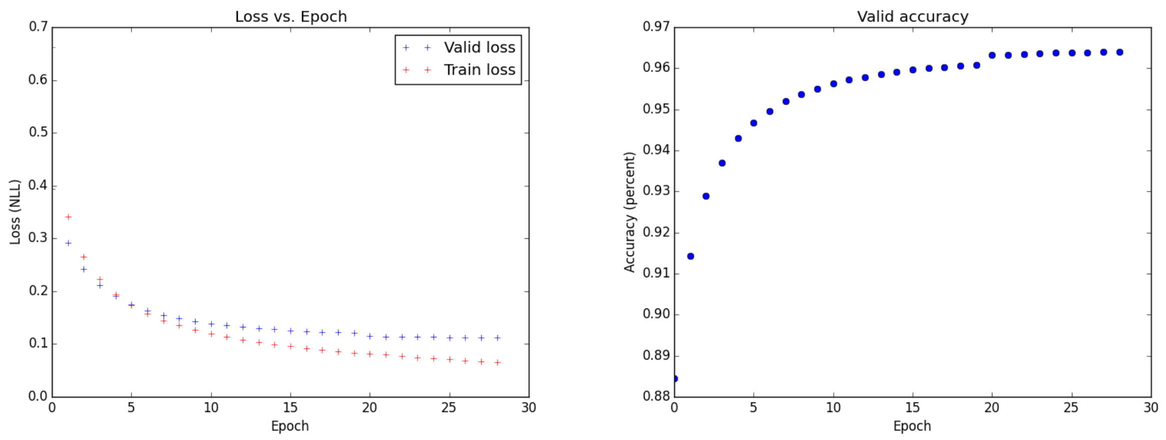

Figure 2: Training curves.

Table 2 gives the classification accuracy of each model on the validation set. The results shown are after training 20 epochs, except for models with Glove embeddings (which we trained for longer). For neural models, GLOVE means we used pretrained Glove word embeddings and SUFF means we included suffix features.

Model	Acc.
NAIVE BAYES	91.27
NAIVE BAYES (NO CAP FEATURES)	89.98
LOGISTIC REGRESSION	76.50
NEURAL	92.11
NEURAL+GLOVE	96.41
NEURAL+GLOVE+SUFF	96.60

Table 2: Results on PTB POS tagging.

For naive Bayes, we tried including and excluding the capitalization features, and it turns out that capitalization gives better accuracy as expected. The model with suffixes included does the best, suggesting that more features should be experimented with to retrieve more signal.

As an extension, we also tried to run Adadelta instead of SGD. However, because it saves $O(P)$ values for each iteration, where P is the number of parameters in the model, it turns out to be prohibitively expensive for memory, and training did not finish in time.

5 Conclusion

In this homework, we built a POS tagger by preprocessing the PTB text corpus into window-sized data points. Then we ran baseline models of naive Bayes and logistic regression, and finally applied the neural network model of Collobert et al (2011).

We found that training these neural models takes a significant amount of time, and thus careful logging and saving models after each epoch was important, allowing us to restart training at any checkpoint. Running on Odyssey also helped by offloading computation to a cloud server. We expect that with GPUs, training these models will be much quicker and easier.

Also since restarting training with a new batch shuffle seems to change validation accuracy fairly significantly, we should probably shuffle the batches each time.

The code for this homework can be found here: `https://github.com/r0hilp/cs287-hw2`