HW2: Tagging from Scratch

Virgile Audi
Kaggle ID: Virgile Audi
vaudi@g.harvard.edu

Nicolas Drizard
Kaggle ID: nicodri
nicolasdrizard@g.harvard.edu

February 22, 2016

1 Introduction

This assignement aims to tackle the task of part-of-speech tagging based on the paper by Collobert et al. (2011).

We applied several models, both in terms of the features used and of the models. First, we applied a multi-class Naive Bayes and then a Multinomial Logistic Regression. We used slicing window features with information from the words and capitalization. We tried both with and without the position of the words inside the window.

Then, we used a neural network architecture with two layers. We ran several experiments on this model which was the most accurate. For instance, we trained it on a pre-trained embeddings of words from Pennington et al. (2014). This lead to our best prediction on the test set.

http://github.com/virgodi/cs287

This repository also contains iTorch notebooks where we drafted code.

2 Problem Description

The problem to solve is mult-class classification of tags on words, aka part-of-speech tagging based.

We have a training set of around 600 000 words in sentences, and a validation and test set of both around 100 000 words. We pre-processed them to extract in slicing windows of 5 words a vector of word-index and a vector of capitalisation feature for each word. We followed the conventions used by Collobert et al. (2011). The words-index were extracted from the dictionary of words of the embeddings. For the capitalisations features, we used the following values:

- 1: lower case words
- 2: capital letter feature
- 3: first letter in caps
- 4: else

The output is a set of 45 classes of tags. We evaluated our model on the validation set to tune the hyperparameters.

3 Model and Algorithms

We present here in more details each model with its specificity and the different algorithms we used.

3.1 Multinomial Naive Bayes

The multinomial Naive Bayes (**?**) is a generative model, it means that we specify the class conditional distribution $p(x|y = c)$ as a multinoulli distribution. The main assumption here, which justifies the 'naive' name, is that the feature are condionnaly independent given the class label.

The goal is then to select the parameters that maximizes the likelihood of the training data:

$$\boldsymbol{p(y = \delta(c)) = \sum_{i=1}^{n} \frac{\mathbf{1}(y_i = c)}{n}}$$

We also define the count matrix $\boldsymbol{F}$ to compute the closed form of the probability of $\boldsymbol{x}$ given $\boldsymbol{y}$,

$$F_{f,c} = \sum_{i=1}^{n} \mathbf{1}(\boldsymbol{y}_i = c)\mathbf{1}(x_{i,f} = 1) \text{ forall } c \in \mathcal{C}, f \in \mathcal{F}$$

Then,

$$p(x_f = 1|\boldsymbol{y} = \delta(c)) = \frac{F_{f,c}}{\sum\limits_{f' \in \mathcal{F}} F_{f',c}}$$

Knowing these parameters we can compute the probabity of the class given the features:

$$p(\boldsymbol{y} = c|x) \propto p(\boldsymbol{y} = c) \prod_{f \in \mathcal{F}} p(x_f = 1|\boldsymbol{y} = \delta(c))$$

We can add a hyperparameter to handle the long tail of words by distributing the means. We add a Laplacian smoothing parameter α as follows:

$$\hat{\boldsymbol{F}} = \alpha + F$$

3.2 Multinomial Logistic Regression

As a second baseline model, we implement a multiclass logistic regression for this problem. We won't recall the theoretical framework from homework 1 here, but will explain how it applies to speech tagging.

For this particular problem of speach tagging, we had the two same set of input features as for the Naive Bayes approach, i.e. the word windows as well as the capital windows. The main difference however is that for fitting this particular model, we needed to create some new features based on the position of the word and the type of capitalisation in the window. To do so we indexed words in first position from 1 to 100 002 (the size of the vocabulary), words in second position from 100 003 to 200 004, etc. For instance if the first training point was originally represented by:

$$\begin{pmatrix}1 & 1 & 5032 & 2 & 4\end{pmatrix},$$

it was now inputed as:

$$\begin{pmatrix}1 & 100\,003 & 205\,036 & 300\,008 & 400\,014\end{pmatrix}$$

In total, we needed to fit 100 002x5 = 500 010 parameters. After writing a functional explicit stochastic gradient descent algorithm, we decided to use the more optimised function **nn.StochasticGradient()** in order to gain some valuable training time. This function, for which documentation can be found on here (hyperlink), takes for input a dataset with a special table structure, the model itself and the criterion. We can also tune the learning rate and number of epochs to run. It is not clear though what the size of the minibatch is but the function was fast enough not to worry about it.

As the primary goal of this homework was building the neural network, we did not intensively work on the Logistic Regression and kept it rather simple, only implementing a version that only took into consideration the word windows. We will present the results in the Experiments section.

3.3 Neural Network Model

We then focused on the primary objective of the homework which was building a neural network for word tagging. The model is designed as follows.

We input both word and cap windows, both of dimension 5. Each of these windows are then passed through look-up tables of dimensions dim(Vocabulary) x (hidden-parameter) and dim(Cap) x (hidden-parameter) . For each window, this outputs a 5xhidden-parameter tensor. We then concatenate corresponding tensors for a particular window to form a vector of dimension 1x(10∗hidden-parameter).

We then applied a linear transformation to this vector followed by a layer to extract highly non-linear features in the form of a hardtanh() module. The output of this hardtanh() module is then passed through an other linear layer to output a score for each of the 45 possible tags, followed by a softmax() module to transform these scores into a probability distribution over these tags.

This design therefore introduces to hidden parameters that need to be tuned, the dimension of the look-up table and the dimension of the output of the first Linear layer. We will present various experiments below.

In order to simplify the code and be able to use the nn.StochasticGradient() function to train the neural network and taking into account that it only takes a single datastructure argument, we chose to reformat the input data. Instead of having to deal with two different input tensors, i.e. the word and caps tensors, we concatenated the into one single tensor of size: dim(train) x 10. To

be able to then use a single Look-up table, we had to index the caps from 100 003 to 100 006 and count them as extra words.
We would now like to share a trick that helped us run the code. When first coded the Sequential module in Torch, we had issues passing from the concatenation step to the first linear layer due to dimension issues in minibatching. Debugging the code, we noticed that the outputed tensor from the concatenation had an extra unnecessary 1 dimension that came from applying the nn.View() layer. As suggested in the following discussion tread , we implemented a nn.Squeeze() layer similar to the nn.View() in order to transfer what the torch.Squeeze() function did to remove unwanted 1D dimensions in a tensor.

The url of the group is: https://groups.google.com/forum/#!topic/torch7/u4OEc0GB74k

We now present results of the experiments.

4 Experiments

We applied our three models on the Stanford Sentimental dataset and report in a table below our results. We show the running time to emphasize how faster is the Naive Bayes and the accuracy both on the training and test set. We also show the loss, its the exact value for the Naive Bayes and an approximation on the last mini-batch of the epoch (extrapolated then to the whole dataset) for the two other models.

We ran a validation pipeline to come up with the best set of parameters. We also coded a k-fold cross validation but due to a lack of time, did not experiment enough to show interesting results. We therefore retained the set of parameters using validation which optimizes the accuracy. We kept the same seed and the exact other same parameters for the different models training. We obtained the following parameters for each model:

- **Naive Bayes** $\alpha = 1$
- **Logistic Regression** Batch size $= 50, \quad \eta = 1, \quad \lambda = 0.1$
- **Linear SVM** Batch size $= 50, \quad \eta = 1, \quad \lambda = 0.1$

If we look at the results below, we can note that Naive Bayes has the highest Training accuracy but smallest Test accuracy, which seems to indicate that Naive Bayes might be a slightly more overfitting algorithm than the other two. We report the accuracy on the three dataset: train, validation and test (from our Kaggle submission).

Variances of the outputs of these algorithms are also key insights for analysis.

What we can see is that parametrisation is much more crucial for the Naive Bayes algorithm, with performance almost increased by 50% by adding the smoothing paramater $\alpha = 1$. Logistic Regression and linear SVM have very similar performance, both on accuracy and runtime. On the other hand, the Naive Bayes algorithm runs way faster and this aspect must be taken into account when having to choose an algorithm to run.

Table 1: Results Summary

	Training	Validation	Test	Run Time	Loss
Naive Bayes	0.666	0.399	0.344	5-6s	XX
Logistic Regression	0.601	0.403	0.354	85-87s	4x10^{12}
Linear SVM	0.631	0.411	0.350	86-90s	1.21x10^{5}

Table 2: Range of prediction accuracy

	Min Accuracy	Max Accuracy
	Validation	Validation
Naive Bayes	0.257	0.399
Logistic Regression	0.367	0.403
Linear SVM	0.333	0.411

5 Conclusion

This assignement made us build three different linear models for a text classification task. Whereas the naive bayes is fast to train, the linear SVM and the logistic regression require an optimization algorithm (here stochastic gradient descent). However, the accuracy reached are pretty similar for the different models. We also realized the importance of the tuning part on the hyperparameters to improve our classification accuracy.

There is still room for improvement if more time. First, we could build more features on our reviews. In the current models, we just considered each word separately and built a one-hot feature from it. We could also consider the sequence of consecutive words of a given size n, called also n-grams, to incorporate the relationships between the words to our features. We could also think of additional linear models.

References

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. volume 12, pages 2493–2537. JMLR.org.

Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.