HW1: Classification

Roshan Padaki
rpadaki@college.harvard.edu

Michael Zhang
michael_zhang@college.harvard.edu

February 7, 2019

1 Introduction

Sentence classification is a widely studied problem in natural language processing (NLP) (**?**, **?**). Here we focus on text classification, the task of identifying positive or negative sentiment on a given sentence. Many methods exist for building classifiers, and although NLP has enjoyed its fair share of success due to recent advances in deep learning and more generally neural networks (**?**), we seek to compare how exactly these models stack up against others.

Towards this end, we benefit from two developments: word embeddings and convolutional neural networks (CNNs) for NLP. As elucidated in **?**, much work in deep learning for NLP relies on the learning word representations through language models. Known commonly as embeddings, these functions can be trained on large corpuses of text to map from sparse one-hot encoding representations spanning an entire vocabulary size to a lower dimensional vector space that effectively encodes semantic features along its dimensions. Towards sentiment analysis, such denser representations are intuitively helpful for classifying different words through their inferred underlying semantics, as similar words are geometrically closer in the encoded vector space.

CNNs, although originally invented for computer vision (**?**), have enjoyed success in NLP as an alternative to traditional methods heavy on linguistic expertise and feature-engineering (**?**). Briefly, CNNs can make use of various filters convolved over some initial word representation to learn features locally in a hidden layer. More so, these local features can be passed through as inputs to a neighboring hidden layer, and multiple layers can be chained together to enable more complex feature learning.

For this assignment, we train four main model families: Naive Bayes, Logistic Regression, Continuous Bag-of-Words (CBOW), and CNNs, and compare their accuracies for predicting sentiment on the Stanford SST-2 dataset. Additionally, we make use of static pre-trained word vectors courtesy of fastText trained on Wikipedia. (source). Finally, we explore improvements through tuning parameter and trying out different model architectures.

2 Problem Description

From the Stanford sentiment data, we aim to build four classes of classifiers that reliably predict sentiment given an input sentence. Each sentence has been tagged either positive or negative, removing all neutral sentiment. Accordingly, each classifier aims to learn a probability distribution $p(\boldsymbol{y}|\boldsymbol{x})$ where $y_i \in \boldsymbol{y}$ is a sentiment label such that $y_i \in \{0, 1\}$ for all data indices i (as according

to the binary labelled SST-2 dataset) and $\boldsymbol{x}_i \in \boldsymbol{x}$ is a feature vector, derived from a given sentence. Our predictions y for test case j are then of the form

$$y_j = \sigma(\boldsymbol{W}^T \boldsymbol{x}_j + b)$$

where σ is our activation function, $\boldsymbol{W}$ is a matrix of learned weights, $\boldsymbol{x}$ is the feature vector for input j, and b is a bias vector.

Beyond this general model formulation, we also define the following variables which will be useful to reference in the next section.

Variable	Definition
σ	Activation function (sigmoid, tanh, ReLU)
$\boldsymbol{b}$	Bias term(s)
$\boldsymbol{e}$	Embedding vector
$\boldsymbol{x}$	Word vector
$\boldsymbol{w}, p, c$	Convolutional terms: filter, dropout prob., feature
$\mathcal{P}, \mathcal{N}$	Training data: positive sentences, negative sentences

3 Model and Algorithms

3.1 Naive Bayes

Our first classifier trains weights $\boldsymbol{W}$ and bias $\boldsymbol{b}$ as follows, to make a prediction

$$y = \sigma\left(\boldsymbol{W}^{\mathsf{T}} \phi_{\max}(\boldsymbol{x}_{1:t})\right)$$

Given a smoothing parameter α, we create a mollified set-of-words feature count $\boldsymbol{p}$ indexed by $\mathcal{V}$, so that for $\mathcal{P}$ the positive sentences,

$$\boldsymbol{p}_f = \alpha + |\{\boldsymbol{x} \in \mathcal{P} : f \in \boldsymbol{x}\}|\,.$$

Similarly, we get a count

$$\boldsymbol{q}_f = \alpha + |\{\boldsymbol{x} \in \mathcal{N} : f \in \boldsymbol{x}\}|\,.$$

Using these counts, we set our weight vector to be

$$\boldsymbol{w} = \log\left(\frac{p/\|p\|_1}{q/\|q\|_1}\right).$$

Meanwhile, we set our bias $\boldsymbol{b}$ as $\log(|\mathcal{P}|/|\mathcal{N}|)$. To interpret the smoothing parameter, we see that as $\alpha \to \infty$, $\boldsymbol{W} \to 0$, whence our probability is exactly $y = \sigma(0) = 0.5$.

In this sense, α can be understood as a noise-smoothing parameter. For Naive Bayes, we followed **?**, and set $\alpha = 1$.

3.2 Logistic Regression

We learn weights $\boldsymbol{w}$ and bias $\boldsymbol{b}$ to optimize prediction

$$y = \sigma\left(\sum_i \boldsymbol{W}^\mathsf{T} \boldsymbol{x}_i + \boldsymbol{b}\right)$$

where $\boldsymbol{x}_i$ is a $|\mathcal{V}|$-dimensional vector representing bag-of-words unigram counts for each training sample, and σ is the standard sigmoid activation function

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

We implement the weight and bias matrices as a single fully-connected layer in PyTorch, mapping to a two element output under our activation function.

3.3 Continuous Bag-of-Words

In our continuous bag-of-words (CBOW) model, each word is an n-length sentence is first mapped to an embedding vector $\boldsymbol{e}_i$ and all embedding vectors are then averaged to produce a single feature vector $\boldsymbol{e}$ to represent the entire input. Like the regular bag-of-words model, the order of words as they appear in each training sentence is not taken into account, but this mapping uses a continuous distribution to represent context of a sentence. Accordingly, we have

$$\boldsymbol{e} = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{e}_i$$

which is then fed into a softmax-activated fully-connected layer.

3.4 Convolutional Neural Nets

Finally, our basic implementation for a CNN follows that of **?**. Let $\boldsymbol{x}_i \in \mathbb{R}^k$ be the k-dimensional word vector which corresponds to the i-th word of a n-word long sentence. Accordingly, such a sentence is represented as

$$\boldsymbol{x}_{1:n} = \boldsymbol{x}_1 \oplus \boldsymbol{x}_2 \oplus \ldots \oplus \boldsymbol{x}_n$$

where $\oplus$ is the concatenation operator. Given this representation, we can generate new features c_i by applying a filter $\boldsymbol{w} \in \mathbb{R}^{hk}$ to concatenations of h words, or windows. Accordingly,

$$c_i = \sigma(\boldsymbol{w} \cdot \boldsymbol{x}_{i:i+h-1} + b)$$

where σ is a nonlinear function such as Tanh or ReLU, $\boldsymbol{x}_{i:i+h-1}$ is a window with h words, and $b \in \mathbb{R}$ is the bias term. The filter is applied to each possible window of words in a sentence to produce a feature map

$$\boldsymbol{c} = [c_1, c_2, \ldots, c_{n-h+1}]$$

Following **?** and **?**, we apply a max-over-time pooling operation over $\boldsymbol{c}$ and take $c = \max(\boldsymbol{c})$ to be the feature corresponding to the filter.

To obtain multiple features, we repeat the process for multiple filters, ultimately passing them to a fully connected softmax layer with binary outputs for our prediction. In our PyTorch implementation, we fine-tune our word vectors via backpropagation, and employ dropout according to **?** with $p = 0.5$ for regularization to avoid overfitting.

4 Experiments

For all models, we trained with 30 epochs on batches of size 30 unless otherwise noted. In addition to the Stanford SST-2 dataset supplied by torchtext, we implement all models as "named" versions using NamedTensor. As noted on Kaggle, we compare our submissions both with each other and a single class baseline, which achieves a public dataset testing accuracy of 52.38%. Our basic results are listed in Table 1.

Model	Accuracy (%)
SINGLE CLASS	52.38
NAIVE BAYES	82.15
LOGISTIC REGRESSION	78.41
CBOW	77.05
CNN	79.49

Table 1: Basic model performance

Although not actually implemented, we note that the single class baseline model does not perform well, with all other models exhibiting notable performance gains. Notably, although we did not expect Naive Bayes to be the top performer given its strong assumptions on input independence, our results show otherwise.

Given the successful performance of CNNs in **?**, we were interested in experimenting with the parameters further. Our main objective was to try to reproduce the 87.2 % accuracy reported on the SST-2 dataset. Accordingly, using default training parameters (filter lengths $2, 3, 4$; number of filters 100 learning rate $2e^{-4}$; batch size 10; dropout 0.5), we experimented with stride length, filter lengths, hidden layer depth, and an alternate pre-trained embedding (GloVe). However, our results (summarized in Table 2) do not show clear improvements.

Model Modification	Accuracy (%)
BASELINE	79.49
STRIDE LENGTH	80.07
FILTER LENGTHS	78.41
EMBEDDING	81.49

Table 2: Modified CNN model performance. ***Stride Length****: Modification to the stride length from default length* 1*, with optimal performance at length* 2*.* ***Filter Lengths****: Adding a filter with size* 2 *to the default filters of size* $3, 4, 5$*.* ***Embedding****: Building the vocab representation with Stanford's GloVe embedding*

.

5 Conclusion

We built and trained different models to classify the SST data set, with overall success compared to the baseline. Surprisingly, the most successful model was Naive Bayes, which strongly assumes that features are independent of each other within a class. This may indicate that, in this data set,

features more strongly correlated with a class actually tend to occur more independently of each other.

Nevertheless, all of our models performed quite well. Although we were not able to replicate the CNN performance exhibited in **?**, we reached similar conclusions that with little hyperparameter tuning, a simple CNN with one layer of convolution and pre-trained embeddings worked well. In fact, the rather close performance of all model classes dependent on the our embedding, and the performance boost in CNNs seen when switching from the default wiki-based vectors to GloVe, further corroborate the idea that advances in NLP can be associated with unsupervised pretraining of word vectors.

The biggest difficulty we had throughout this process was acquainting ourselves with NamedTensor and sorting out bugs in our code resulting from the new package. As a whole, we found the use of named dimensions to be very useful, but had to spend a bit more time wrangling and managing our data.

Our code can be found at `https://github.com/rpadaki/cs287assignments/tree/master/hw1`.