Linear and Neural Sentiment Classification

CSE 5525: Assignment 1

Goals

The main goal of this assignment is to get experience with linear classifiers and simple neural networks, comparing and contrasting how the two perform on sentiment analysis. You will see the standard pipeline used in many NLP tasks (reading in data, preprocessing, training, and testing), understand how feature extraction works for linear models, and see how that compares to a shallow neural network based on word embeddings. **Your task is to complete both parts below, as well as TWO of the optional "Exploration" pieces.**

Dataset and Code

Please use `Python 3.5` and a recent version of PyTorch for this project.

Installing PyTorch: You will need PyTorch for this project. To get it working on your own machine, you should follow the instructions on this link. The assignment is small-scale enough to complete using CPU only, so don't worry about installing CUDA and getting GPU support working unless you want to.

Data: You'll be using the movie review dataset of Socher et al. [2013]. This is a dataset of movie review snippets taken from Rotten Tomatoes. The labeled data actually consists of full parse trees with each constituent phrase of a sentence labeled with sentiment (including the whole sentence). The labels are "fine-grained" sentiment labels ranging from 0 to 4: highly negative, negative, neutral, positive, and highly positive. We are tackling a simplified version of this task which frequently appears in the literature: positive/negative binary sentiment classification of sentences, with neutral sentences discarded from the dataset. The data files given to you contain newline-separated sentiment examples, consisting of a label (0 or 1) followed by a tab, followed by the sentence, which has been tokenized but not lowercased. The data has been split into a train, development (dev), and a blind test set. On the blind test set, you do not see the labels and only the sentences are given to you. The framework code reads these in for you.

Getting Started: Download the code and data. Expand the file and change into the directory. To confirm everything is working properly, run:

```
python sentiment_classifier.py --model TRIVIAL
```

This loads the data, instantiates a `TrivialSentimentClassifier` that always returns 1 (positive), and evaluates it on the training and dev sets. The reported dev accuracy should be Accuracy: 444/872 = 0.509174. Always predicting positive isn't so good!

Framework code: The code you are given consists of several files. `sentiment_classifier.py` is the main class. It uses `argparse` to read in several command line arguments. It's okay to add command line arguments during development or do whatever you need for experimentation, but your submission to Gradescope should work without modifying this file. For the existing arguments, you should not need to modify the paths if you execute within the `p1-distrib` directory. `--model`

and `--feats` control the model specification. This file also contains evaluation code. The main method loads in the data, initializes the feature extractor, trains the model, and evaluates it on train, dev, and blind test, and writes the blind test results to a file. Data reading is handled in `sentiment_data.py`. This also defines a `SentimentExample` object, which wraps a list of words with an integer label (0/1), as well as a `WordEmbeddings` object, which contains pre-trained word embeddings for this dataset. `utils.py` implements an `Indexer` class, which can be used to maintain a bijective mapping between indices and features (strings). `models.py` is the primary file you will be modifying. It defines base classes for the `FeatureExtractor` and the classifiers, and defines the train logistic regression and train deep averaging network functions, which you will be implementing. `train_model` is your entry point which you may modify if needed.

Part 1: Logistic Regression (30 points)

In this part, you should implement a logistic regression classifier with a bag-of-words unigram featurization, as discussed in lecture and the textbook. This will require modifying train logistic regression, `UnigramFeatureExtractor`, and `LogisticRegressionClassifier`, all in `models.py`. `train_logistic_regression` should handle the processing of the training data using the feature extractor. `LogisticRegressionClassifier` should take the results of that training procedure (model weights) and use them for inference. **The purpose of this assignment is to train logistic regression "from scratch" rather than using scikit-learn. Accordingly, the autograder does not have sklearn installed.**

Feature Extraction: First, you will need a way of mapping from sentences (lists of strings) to feature vectors, a process called feature extraction or featurization. A unigram feature vector will be a sparse vector with length equal to the vocabulary size. There is no single right way to define unigram features. For example, do you want to throw out low-count words? Do you want to lowercase? Do you want to discard stopwords? Do you want the value in the feature vector to be 0/1 for absence or presence of a word, or reflect its count in the given sentence? You can use the provided `Indexer` class in `utils.py` to map from string-valued feature names to indices. Note that later in this assignment when you have other types of features in the mix (e.g., bigrams in Part 3), you can still get away with just using a single Indexer: you can encode your features with "magic words" like "Unigram=great" and "Bigram=great—movie". This is a good strategy for managing complex feature sets. There are two approaches you can take to extract features for training: (1) extract features "on-the-fly" during training and grow your weight vector as you add features; (2) iterate through all training points and pre-extract features so you know how many there are in advance (optionally: build a feature cache to speed things up for the next pass).

Feature Vectors: Since there are a large number of possible features, it is always preferable to represent feature vectors sparsely. That is, if you are using unigram features with a 10,000 word vocabulary, you should not be instantiating a 10,000-dimensional vector for each example, as this is very inefficient. Instead, you want to maintain a list of only the nonzero features and their counts. Our starter code suggests Counter from the collections as the return type for the extract features method; this class is a convenient map from objects to floats and is useful for storing sparse vectors like this.

Weight Vector: The most efficient way to store the weight vector is a fixed-size numpy array.

SGD and Randomness: Throughout this course, the examples in our training sets not necessarily randomly ordered. You should make sure to randomly shuffle the data before iterating through it.

Even better, you could do a random shuffle every epoch.

Random Seed: If you do use randomness, you can either fix the random seed or leave it variable. Fixing the seed (with random.seed) can make the behavior you observe consistent, which can make debugging or regression testing easier (e.g., ensuring that code refactoring didn't actually change the results). However, your results are not guaranteed to be exactly the same as in the autograder environment.

Part 1 Deliverable Implement unigram logistic regression. To receive full credit on this part, you must get at least 77% accuracy on the development set, and the training and evaluation (the printed time) should take less than 20 seconds[1].

Exploration: Try a few different "schedules" for the step size (having one be the constant schedule is fine). One common one is to decrease the step size by some factor every epoch or few; another is to decrease it like $1/t$. How do the results change? Compare the training accuracy and development accuracy of the model. Plot (using matplotlib or another tool) the training objective (dataset log likelihood) and development accuracy of logistic regression vs. number of training iterations for a couple of different step sizes. Think about what you observe and what it means.

Exploration: Implement and experiment with `BigramFeatureExtractor`. Bigram features should be indicators on adjacent pairs of words in the text. Then experiment with at least one other feature modification in `BetterFeatureExtractor`. Things you might try: other types of n-grams, tf-idf weighting, clipping your word frequencies, discarding rare words, discarding stopwords, etc. This feature modification should not just consist of combining unigrams and bigrams.

Part 2: Deep Averaging Network (40 points)

In this part, you will implement a deep averaging network as discussed in lecture and in Iyyer et al. [2015]. If our input is s = (w1, . . . , wn), then we use a feedforward neural network for prediction with input $1/n \sum_{i=1}^{n} e(w_i)$ where e is a function that maps a word w to its real-valued vector embedding. train deep averaging network is your entry point for this part. You will now need the `WordEmbeddings` class. This class wraps a matrix of word vectors and an Indexer in order to index new words. The Indexer contains two special tokens: `PAD` (index 0) and `UNK` (index 1).

`UNK` can stand in words that aren't in the vocabulary, and `PAD` is useful for implementing batching later. Both are mapped to the zero vector by default. You will want to use

`get_initialized_embedding_layer` to get a `torch.nn.Embedding` layer that can be used in your network. This layer is trainable (if you set frozen to False, which will be slower), but is initialized with the pre-trained embeddings.

Data: You are given two sources of pretrained embeddings you can use:

`data/glove.6B.50d-relativized.txt` and `data/glove.6B.300d-relativized.txt` (the default, which usually works better but takes longer). The loading of this is controlled by the

`--word_vecs_path`. These are trained using **GloVe** [Pennington et al., 2014]. The vectors have been relativized to your data, meaning that they do not contain embeddings for words that don't occur in the train, dev, or test data. This is purely a runtime and memory optimization.

[1] Our autograder hardware is fairly powerful and we will be lenient about rounding, so don't worry if you're close to this threshold.

PyTorch Example: `ffnn_example.py` implements the network discussed in lecture for the synthetic XOR task. It shows a minimal example of the PyTorch network definition, training, and evaluation loop. Feel free to refer to this code extensively and to copy-paste parts of it into your solution as needed. Most of this code is self-documenting. The most unintuitive piece is calling zero grad before calling backward! Backward computation uses in-place storage and this must be zeroed out before every gradient computation.

Implementation: Following the example, the rough steps you should take are:

1. Define a subclass of `nn.Module` that does your prediction. This should return a log-probability distribution over class labels. Your module should take a list of word indices as input and embed them using a `nn.Embedding layer` initialized appropriately.
2. Compute your classification loss based on the prediction. In lecture, we saw using the negative log probability of the correct label as the loss. You can do this directly, or you can use a built-in loss function like `NLLLoss` or `CrossEntropyLoss`. Pay close attention to what these losses expect as inputs (probabilities, log probabilities, or raw scores).
3. Call `network.zero_grad()` (zeroes out in-place gradient vectors), `loss.backward` (runs the backward pass to compute gradients), and `optimizer.step` to update your parameters.

Implementation and Debugging Tips: Come back to this section as you tackle the assignment!

- You should print training loss over your models' epochs; this will give you an idea of how the learning process is proceeding.
- You should be able to do the vast majority of your parameter tuning in small-scale experiments. Try to avoid running large experiments on the whole dataset in order to keep your development time fast.
- If you see NaNs in your code, it's likely due to a large step size. `log(0)` is the main way these arise.
- For creating tensors, `torch.tensor` and `torch.from_numpy` are pretty useful. For manipulating tensors, permute lets you rearrange axes, squeeze can eliminate dimensions of length 1, expand or repeat can duplicate a tensor across a dimension, etc. You probably won't need to use all of these in this project, but they're there if you need them. PyTorch supports most basic arithmetic operations done element-wise on tensors.
- To handle sentence input data, you typically want to treat the input as a sequence of word indices. You can use `torch.nn.Embedding` to convert these into their corresponding word embeddings.
- Google/Stack Overflow and the PyTorch documentation are your friends. Although you should not seek out prepackaged solutions to the assignment itself, you should avail yourself of the resources out there to learn the tools.

Part 2 Deliverable: Implement the deep averaging network. Your implementation should consist of averaging vectors and using a feed-forward network, but otherwise you do not need to exactly reimplement what's discussed in Iyyer et al. [2015]. Things you can experiment with include varying the number of layers, the hidden layer sizes, which source of embeddings you use (50d or 300d),

your optimizer (Adam is a good choice), the nonlinearity, whether you add dropout layers (after embeddings? after the hidden layer?), and your initialization.

You should get least 77% accuracy on the development set without the autograder timing out; you should aim for your model to train in less than 10 minutes or so (and you should be able to get good performance in 3-5 minutes on a recent laptop).

If the autograder crashes but your code works locally, there is a good chance you are taking too much time (the autograder VMs are quite weak). Try reducing the number of epochs so your code runs in 3-4 minutes and resubmitting to at least confirm that it works.

Exploration: Implement batching in your neural network. To do this, you should modify your `nn.Module` subclass to take a batch of examples at a time instead of a single example. You should compute the loss over the entire batch. Otherwise your code can function as before. You can also try out batching at test time by changing the `predict_all` method. Note that different sentences have different lengths; to fit these into an input matrix, you will need to "pad" the inputs to be the same length. If you use the index 0 (which corresponds to the PAD token in the indexer), you can set `padding_idx=0` in the embedding layer. For the length, you can either dynamically choose the length of the longest sentence in the batch, use a large enough constant, or use a constant that isn't quite large enough but truncates some sentences (will be faster).

Exploration: Try out an additional neural architecture, such as an LSTM or a CNN. What hyperparameters did you use? What performance do you observe?

Writeup (30 points) and Code Submission

Writeup: You should submit a brief 1-page writeup. In it, you should describe anything distinctive about your implementation, but **mostly focus on reporting your results and describing what you tried with the two Exploration pieces**. Your writeup should be at most 1 page of text, with a second page for any plots or graphs necessary. (You will not be penalized for including more, but we do not want to see long writeups for this part.)

Code Submission: TBD

References

Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. Deep unordered composition rivals syntactic methods for text classification. In Chengqing Zong and Michael Strube, editors, *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1681–1691, Beijing, China, July 2015. Association for Computational Linguistics. doi: 10.3115/v1/P15-1162. URL `https://aclanthology.org/P15-1162/`.

Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1162. URL `https://aclanthology.org/D14-1162/`.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In David Yarowsky, Timothy Baldwin, Anna Korhonen, Karen Livescu, and Steven Bethard, editors, *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA, October 2013. Association for Computational Linguistics. URL `https://aclanthology.org/D13-1170/`.