Natural Language Processing
Assignment 2: Text Classification

1 Implementation/Experimentation with Linear Text Classifiers (50 points)

You will implement and experiment with simple ways of building text classifiers based on linear models. Consider a classifier defined by the function classify:

$$\text{classify}(\boldsymbol{x}, \mathbf{w}) = \underset{y \in \mathcal{L}}{\text{argmax}} \ \ \text{score}(\boldsymbol{x}, y, \mathbf{w})$$

where $\boldsymbol{x}$ is a textual input, y is an output class label, $\mathcal{L}$ is the space of all possible classification labels, and the parameters are contained in the parameter (weight) vector $\mathbf{w}$. The score function is defined:

$$\text{score}(\boldsymbol{x}, y, \mathbf{w}) = \sum_i w_i f_i(\boldsymbol{x}, y)$$

where each f_i is a feature function and w_i is the corresponding weight of the feature function.

Provided Data:

- `sst3.zip`: contains `sst3.{train,dev,devtest,train-full-sentences}`; 3-way sentiment analysis of movie reviews, adapted from the 5-way dataset from Socher et al. (2013). Each line in each file contains a textual input followed by a tab followed by an integer containing the gold standard label. The label set $\mathcal{L}$ is $\{0, 1, 2\}$, where 0 is negative sentiment, 1 is neutral, and 2 is positive sentiment. Note that the `train` file has some lines that are complete sentences and other lines that are constituents within sentences, while `dev` and `devtest` contain only complete sentences. The file with suffix `train-full-sentences` contains only the complete sentences from the `train` file. You do not need to use `train-full-sentences`. Training on `train` usually works better. We provide `train-full-sentences` since you may want to refer to it when designing features or doing analysis.

1.1 Building a Linear Classifier (24 points)

Implement a linear model for 3-way sentiment classification using the features and loss function specified below.

Learning:

For learning, use the **perceptron loss function**:

$$\text{loss}_{\text{perc}}(\boldsymbol{x}, y, \mathbf{w}) = -\text{score}(\boldsymbol{x}, y, \mathbf{w}) + \max_{y' \in \mathcal{L}} \text{score}(\boldsymbol{x}, y', \mathbf{w})$$

We will denote our training dataset by $\mathcal{T} = \{\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle\}_{i=1}^{|\mathcal{T}|}$, where $y^{(i)} \in \mathcal{L}$ is the label of $\boldsymbol{x}^{(i)}$. We want to minimize the perceptron loss function on the training set $\mathcal{T}$, i.e.:

$$\hat{\mathbf{w}} = \operatorname*{argmin}_{\mathbf{w}} \sum_{i=1}^{|\mathcal{T}|} \text{loss}_{\text{perc}}(\boldsymbol{x}^{(i)}, y^{(i)}, \mathbf{w})$$

We can solve this optimization problem by using stochastic subgradient descent (SSD). The update rule for the above loss for a single weight w_j and a single training example $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$ has the following form:

$$w_j \leftarrow w_j - \eta \frac{\partial \text{loss}_{\text{perc}}(\boldsymbol{x}^{(i)}, y^{(i)}, \mathbf{w})}{\partial w_j}$$

where η is the **step size** or **learning rate**. For a linear model, entry j in the subgradient is:

$$\frac{\partial \text{loss}_{\text{perc}}(\boldsymbol{x}, y, \mathbf{w})}{\partial w_j} = -f_j(\boldsymbol{x}, y) + f_j(\boldsymbol{x}, \text{classify}(\boldsymbol{x}, \mathbf{w}))$$

So, the update rule becomes:

$$w_j \leftarrow w_j + \eta f_j(\boldsymbol{x}^{(i)}, y^{(i)}) - \eta f_j(\boldsymbol{x}^{(i)}, \text{classify}(\boldsymbol{x}^{(i)}, \mathbf{w}))$$

That is, update the weight w_j by adding the value of the feature function f_j applied to the given training example $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$ (scaled by η) and subtract the value of the feature function f_j applied to the given input $\boldsymbol{x}^{(i)}$ paired with the predicted label $\text{classify}(\boldsymbol{x}^{(i)}, \mathbf{w})$ (again scaled by η). If $y^{(i)} = \text{classify}(\boldsymbol{x}^{(i)}, \mathbf{w})$, i.e., if the current classifier is correct, then the terms will cancel for all weights and no update will be made to $\mathbf{w}$ for $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$.

Classifying held-out data:

Once you have an estimate $\hat{\mathbf{w}}$, use it to classify the held-out data (DEV or DEVTEST) by calling $\text{classify}(\boldsymbol{x}, \hat{\mathbf{w}})$ for each $\boldsymbol{x}$ in the held-out dataset. As the evaluation metric, compute the prediction accuracy, that is, the percentage of instances that were classified correctly.

Features:

Implement **unigram binary** features for your text classifier using the following feature template:

$$f^{\text{u,b}}(\boldsymbol{x}, y) = \mathbb{I}[y = \text{label}] \wedge \mathbb{I}[\boldsymbol{x} \text{ contains } \textit{word}]$$

Using a feature count cutoff of 1, instantiate this feature template for all $\langle \text{label}, \textit{word} \rangle$ pairs observed in your training data. That is, create a binary feature for each word in an input $\boldsymbol{x}^{(i)}$ in the training data $\mathcal{T}$, paired with its observed label $y^{(i)}$. Create a weight w for each feature. This is the first thing you should do in your program after reading in the training data. (Hint: When I did this, I found 36,964 features in $\mathcal{T}$, i.e., 36,964 instances of the unigram binary feature template. This also means that my weight vector $\mathbf{w}$ has 36,964 entries in it.)

Experimental details:

Experiment with your implementation. Set the initial values of the weights ($\mathbf{w}$) to zeroes. Run SSD for N epochs over the training set ($N = 20$ is likely enough), using a mini-batch size of 1 (i.e., "online") and a fixed stepsize of $\eta = 0.01$. An epoch is defined as processing all examples in $\mathcal{T}$. On each epoch, you can simply loop through $\mathcal{T}$ in order.

Periodically during training, compute the classification accuracy on DEV using your current weights. I computed classification accuracy on DEV every 20,000 examples within each epoch, then again at the end of each epoch. Each time your compute classification accuracy on DEV, check if the DEV accuracy is the highest seen so far. If so, compute the classification accuracy on DEVTEST.

After you finish training for N epochs, report the best accuracy achieved on DEV and the accuracy on DEVTEST corresponding to the single model that achieved the best accuracy on DEV. This practice is often called **early stopping** because it correspond to using a model from somewhere in the middle of the training procedure (based on DEV accuracy) rather than using the model at the end of training. **Report your results and submit your code.**

(Hint: my best DEV accuracy tended to be in the range of 64% to 66%, depending on how often I computed DEV accuracy and how many total epochs (N) I trained for. The models with these best DEV accuracies typically reached 55.5% to 56.5% on DEVTEST. Your results may vary, but hopefully they are close to these ranges.)

1.2 Hinge Loss (4 points)

Let's define **hinge loss** as follows:

$$\text{loss}_{\text{hinge}}(\boldsymbol{x}, y, \mathbf{w}) = -\text{score}(\boldsymbol{x}, y, \mathbf{w}) + \max_{y' \in \mathcal{L}} \left(\text{score}(\boldsymbol{x}, y', \mathbf{w}) + \mathbb{I}[y \neq y']\right)$$

with subgradient entry j as follows:

$$\frac{\partial \text{loss}_{\text{hinge}}(\boldsymbol{x}, y, \mathbf{w})}{\partial w_j} = -f_j(\boldsymbol{x}, y) + f_j(\boldsymbol{x}, \text{costClassify}(\boldsymbol{x}, \mathbf{w}))$$

where $\text{costClassify}(\boldsymbol{x}, \mathbf{w})$ is defined:

$$\text{costClassify}(\boldsymbol{x}, \mathbf{w}) = \operatorname*{argmax}_{y' \in \mathcal{L}} \left(\text{score}(\boldsymbol{x}, y', \mathbf{w}) + \mathbb{I}[y \neq y']\right)$$

The SSD update rule for hinge loss can be derived following the same line of reasoning as for the perceptron loss above.

Implement and experiment with minimizing hinge loss instead of perceptron loss. Use the same step size (η) and number of epochs (N) as above. Also as above, compute DEV accuracy periodically during training and compute DEVTEST accuracy when you see a new best accuracy on DEV. **Report your best DEV accuracy and the corresponding DEVTEST accuracy.** Hint: You should see increases in accuracy on DEV and DEVTEST of 1 to 3% compared to using the perceptron loss.

1.3 Feature Weight Analysis (4 points)

Inspect the learned feature weights in the model from 1.2 (i.e., trained with hinge loss) that has the highest accuracy on DEV. **In your report, list the 10 features with the largest weights for each label. Describe what you observe in these features. Is anything noteworthy? How would you characterize the words that appear for each label?**

You can also look at the features with the lowest (i.e., most negative) weights, but they are often harder to interpret. (Hint: as a sanity check, my highest weighted feature for label 0 is the one for the word "lacks" and for label 2 it is the feature for the word "pleasant". Your weights may vary from mine, but hopefully these two words are strongly indicative of these respective labels in your trained model.)

1.4 Error Analysis (8 points)

Inspect the errors made on DEVTEST by the same model from 1.3 above (the model trained with hinge loss that has the highest accuracy on DEV). After looking at a few errors, you will likely notice repeated patterns among them. You should attempt to identify the primary cause of the misclassification by your model. The error could be due to issues with the annotations, challenging linguistic phenomena in the inputs, insufficient training data, inadequate features, overfitting during training, etc. For some of these error categories, you may want to look through your training data and learned model file to identify the cause of the error.

To get you started thinking about error categories, here are some potential categories (some of which we discussed in class):

- incorrect (or at least questionable) gold standard annotation
- negation
- multiple sentiments with most important sentiment at the end
- word sense ambiguity
- unknown words, i.e., words in DEV or DEVTEST but not in the training data, and therefore have no features (for example, the word "treasures" appears in DEVTEST but not in the training data)
- ...

The above list is by no means exhaustive, and you should look for other categories as you go through your model's errors. You can also design subcategories of any of the above categories if you feel they are too general.

In your report, list 20 errors your model made on DEVTEST and, for each, manually choose one error category that fits best. When listing an error in your report, include the sentence, its gold standard label, your model's predicted label, and your annotation of the error category.

1.5 Feature Engineering (10 points)

Hopefully your error analysis gave you lots of ideas for features to add to your model! In this section, you will brainstorm new feature templates, implement them, train using hinge loss, and report the results on DEV and DEVTEST (using the same experimental procedure as used above in 1.2). Hopefully you can find features that both increase your overall DEVTEST accuracy while also addressing some of the error categories you identified above. After learning feature weights for your new features, inspect the learned weights and discuss them in your report: do the feature weights match your expectations or are they different? Looking at the feature weights will help you to determine if the feature templates you designed are helping you to fit the data better in the way that you expected them to.

Define two new feature templates, implement them in your model, and experiment with them. For each feature template, describe it in words (and also formally if that will help us understand it), report the results of running experiments with the new features, and discuss what you observe about their learned weights. You will receive 5 points for each of the two new feature templates you develop (so they should not be trivial variations of each other).

If you impress us by coming up with especially interesting features that lead to significant accuracy gains, you will receive extra credit.

Hints: Remember that your new feature templates can be any function of the entire $\boldsymbol{x}$ and the label y (as long as the function looks at the label y, as we discussed in class). For example, a feature template can look at more than one word in $\boldsymbol{x}$, consider the positions of the words in addition to their presence/absence, count occurrences of words instead of merely checking if they are present, and look at subword structure, among other things. Features can also consider characteristics of the entire sentence (e.g., its length, the number of capitalized words it contains, etc.) and be defined based on external resources like stemmers, lemmatizers, and part-of-speech taggers.

References

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., and Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of EMNLP*. [1]