FNLP Assignment 1 - Sentiment Classification

Ivan Titov, Mirella Lapata

Overview

In this assignment you will learn the basics of tokenization, embeddings, and logistic regression, applied to the task of *sentiment analysis/classification.*

You will build a sentiment classifier from the ground up: You will start by implementing an N-gram (word-level) tokenizer. You will then create features from the tokenized text and build a logistic regression classifier to use these features to perform sentiment analysis. Next, you will write the training loop to train your model on the IMDB dataset. Finally, you will explore a different feature design by using word embeddings to represent the text.

By the end of this assignment you will understand:

- The fundamentals of tokenization: how word-based tokenizers are created and their respective strengths/weaknesses
- How to create count-based features from tokenized text
- How to train a logistic regression model on a binary classification task
- How to compare word embeddings
- How to create document-level features from word embeddings

Total possible points for this assignment: 12 + 20 + 13 + 8 = 53

Submission Guidelines

Your submission deadline is **Friday, 14/02/2025, 12:00PM**.

You will need to submit two files for this assignment:

- A .TAR.GZ file containing your code submission.
- A PDF document with your answers to the questions in this document.

Do not mention any student names in either your code or the PDF.

Code Submission

Your submission file should have the name format "cw1_UUN1_UUN2.tar.gz" where UUN1 and UUN2 refer to the student numbers of the two students who worked on this assignment. The numbers should look like "s1234567".

You should create this file by running (from the parent directory to FNLP_ASSIGNMENT1).

```
$ tar --exclude='data' --exclude='__pycache__' -czvf \
    cw1_UUN1_UUN2.tar.gz fnlp_assignment1/
```

Remember to change the filename to the correct format by replacing UUN1 and UUN2. We recommend unzipping this file and checking the contents to ensure your code is present.

PDF Submission

The PDF submission should have the name format "cw1_UUN1_UUN2.pdf" where UUN1 and UUN2 refer to the student numbers of the two students who worked on this assignment. The numbers should look like "s1234567".

This document should be structured into the same sections as these instructions:

1. Tokenization
2. Logistic Regression
3. Embeddings
4. Looking Forward

At the beginning of the document please add a brief 1-2 sentence statement explaining the contributions of each group member (only refer to group members by UUNs).

Sentiment Analysis

The majority of this assignment will be based on the **sentiment analysis** task, which is defined here as: given a piece of text (in your case, IMDB movie reviews), determine if the text is **positive** or **negative**. For example, the review THIS MOVIE IS THE WORST :(is negative, and the review I LOVE THIS MOVIE! is positive.

The dataset we will be using is the IMDB movie reviews dataset, which contains 50k movie reviews along with sentiment labels. Each datapoint in the dataset is a dictionary with two attributes: `text`, which contains the review text, and `label`, which is either 0 or 1, where 0 indicates negative sentiment and 1 indicates positive sentiment.

Example IMDB review from the dataset (labeled as negative):

its a totally average film with a few semi-alright action sequences that make the plot seem a little better and remind the viewer of the classic van dam films. parts of the plot don't make sense and seem to be added in to use up time. the end plot is that of a very basic type that doesn't leave the viewer guessing and any twists are obvious from the beginning. the end scene with the flask backs don't make sense as they are added in and seem to have little relevance to the history of van dam's character. not really worth watching again, bit disappointed in the end production, even though it is apparent it was shot on a low budget certain shots and sections in the film are of poor directed quality

You might note that reviews are usually rated on a larger scale (e.g. 1-10); the authors of the paper that introduced this dataset converted the raw review scores into binary positive/negative sentiment scores.

Your overarching goal with this assignment will be to train a simple logistic regression classifier to automatically label unseen reviews as positive or negative, using n-gram counts and pretrained word-embeddings.

Programming Setup

To setup your environment, download and unzip **fnlp_assignment1.zip** into your working directory.

Create a conda environment or install the python requirements.

```
$ # to create new conda env
$ conda create -n fnlp python=3.10 --file ./requirements.txt
$ conda activate fnlp
$ # or install into current environment
$ pip install -r requirements.txt
```

Setup the dataset and download word embeddings. **Do not modify the setup_dataset.py script** - it might give you different examples/training splits and result in different performance. More information on downloading the word embedding model is in gensim's documentation), it will require a little over 100MB of space and may take a couple minutes to download:

```
$ python setup_data.py
$ python -m gensim.downloader --download glove-twitter-25
```

To run the unittests:

```
$ python unit_tests.py
```

If you want to run a subset of the tests, you can run:

```
$ python unit_tests.py CLASS
$ python unit_tests.py CLASS.method
```

where CLASS is the name of the class you want to test (LogisticRegressionTest, FeatureExtractorTest, TokenizerTest) and *method* is a method of that class. Loading the word vectors is by far the slowest part of the unit-tests, so we recommend tailoring the command to the part of your code you are trying to evaluate.

Note that success on the unit-tests **does not** guarantee full marks on the assignment or coding sections, don't try to game the unittests.

All of the code you will be changing are within *TODO* and *END TODO* comments, **do not change any other code**. Each section that requires changes will have an exception asking for you to complete that section.

1 Tokenization

The first step in our sentiment classification pipeline is to convert the raw text of the reviews into informative pieces. There are many different ways to do this, we will try a couple and explore their differences.

This step is essential because our model cannot process the raw review text directly - we need to break down the text into individual pieces that we can refer to with unique IDs. The 'pieces' (tokens) we will split text into will be words and short sequences of words called n-grams. We will give each n-gram we see during training a unique ID (called a token ID). The process of going from raw text to these unique IDs is called *tokenization*.

In this part of the assignment you will implement N-gram tokenization.

N-Gram tokenization iterates over the text (in our case, the text will be preprocessed into a list of words), and takes every n-length substring. For us that will look like every n-length sublist of words. For example, the sentence 'I love this movie' with $n = 2$ (a bigram tokenizer) would produce: *(I love), (love this), (this movie)*. Repeating this process across a large corpus (your training set) gives us a set of possible n-grams (e.g. *(love this)*), which we assign unique ids. This process is done via TRAIN_TOKENIZER() method, which calls the TRAIN() method you will implement. Given a new text we can then go backward, and convert each seen n-gram based on its previously assigned id.

For this assignment we will make two small simplifying design decisions: 1) when tokenizing a new text ignore n-grams that didn't appear in your 'training' corpus, they should not correspond to a token id 2) your N-gram tokenizer will ***only*** encode N-grams of that size, not lower. For example, your bigram tokenizer will only encode 2-word combinations, your unigram tokenizer only single words, etc.

Relevant unit-test command:

```
$ python unit_tests.py TokenizerTest
```

You will also apply your tokenizers to a sample text, run the following command for the relevant question:

```
$ python tokenizers.py
```

Tasks (12 points total)

1. Implement the following tokenization methods (**tokenizers.py**) (7 points total):
   - N-gram word level tokenization (e.g., unigram, bigrams, trigrams) in *NgramTokenizer* class (7 points total)
     (a) Implement: TRAIN() - creates token ids for n-grams (4 points)
     (b) Implement: __LEN__() - gets size of vocab (1 point)
     (c) Implement: TOKENIZE() - converts text to token ids (2 point)
2. Apply your Unigram(n=1) and Bigram(n=2) tokenization to the following text (2 points):

   "I love scifi and am willing to put up with a lot. Scifi movies and TV are usually underfunded, under-appreciated and misunderstood."

   - Run **python tokenizers.py** and report the results here: (Note, because your tokenizer is trained on a subset of the training set, it will not have seen all of the n-grams before. Remember to ignore unseen n-grams).

   Report how this text is split into tokens under each method.
3. [1-3 sentences] What are the advantages and disadvantages of n-gram ($n > 1$) vs. unigram tokenization? List at least one advantage and disadvantage. (3 points)

2 Logistic Regression

Now we have a way to split our text into tokens, you will write your own logistic regression model to train on the IMDB dataset.

The first step is to convert your tokenized data into useful features (we will use the n-Gram tokenizer with $n = 2$, i.e. Bigram, for this section). You will implement a count-based feature extractor that, given a text, counts the number of times each token/n-gram appears. The idea behind this featurizer is simple: the more times a word appears, the more likely it is to be relevant to the sentiment of a text.

You will then implement a logistic regression classifier in two parts: a training step that updates the weights+bias from a batch of examples, and a prediction method that uses the models weights+bias to guess a text's sentiment.

The final piece of the puzzle is a training loop: the method that iterates over the training data, updates the model, evaluates the model on the validation set, and at the end returns the best performing model. There are many ways of implementing this training loop so we have broken it into sections for you to complete.

At this point you will be ready to start training your model! We have provided three hyper-parameters in the command-line arguments of **run_classifier.py**: *epochs, learning_rate, and batch_size.* You will try changing these hyper-parameters to boost performance and to get a feel for how these affect training.

Run the final training with:

```
$ python run_classifier.py --model LR --tokenizer NGRAM --ngrams 2 --
    feats COUNTER --learning_rate YOUR_LR --batch_size YOUR_BATCH_SIZE --
    epochs YOUR_EPOCHS
```

And relevant unit-tests with:

```
$ python unit_tests.py CountFeatureExtractorTest
$ python unit_tests.py LogisticRegressionTest
```

Tasks (20 points total)

1. Create useful features from tokenized text (**models.py**) (2 point)
   - Implement *CountFeatureExtractor* class (2 point)
     (a) Implement EXTRACT_FEATURES() - given a text, tokenize and count the occurrence of each token id (2 point)
2. Create Logistic Regression Classifier(**models.py**) (5 points):
   - Implement *LogisticRegressionClassifier* class (5 points)
     (a) Implement PREDICT() - given a text, extract features and compute the model's prediction (2 point)
     (b) Implement TRAINING_STEP() - given a batch of texts, update the weights and bias (3 points)
3. Create training loop (**models.py**) (5 points)
   - Fill in TRAIN_LOGISTIC_REGRESSION() (4 points)
     (a) Instantiate model and tracking variables (1 point)
     (b) Update model from a batch (1 point)
     (c) Get performance metrics at end of epoch (2 point)
     (d) Return best performing model ***by validation accuracy*** (1 point)
4. Train your logistic regression model on IMDB reviews (**python run_classifier.py your-arguments**), using a Bigram tokenizer and the CountFeatureExtractor (3 points).
   - Try different hyperparameters (epochs, lr, batch size) - use the command line arguments instead of hardcoding values
   - Your hyper-parameters must achieve at least 80% accuracy on the test set, and run in under 2 minutes. Report your hyper-parameters and commands to run here (3 points):
5. Create your own custom feature extractor (2 points)
   - Implement CUSTOMFEATUREEXTRACTOR class (just EXTRACT_FEATURES() (2 points)
6. Train your logistic regression model with your custom feature extractor(**python run_classifier.py –feats CUSTOM your-arguments**) (1 points).
   - Try different hyperparameters (epochs, lr, batch size) - use the command line arguments instead of hardcoding values
   - Report your hyper-parameters and commands to run here (1 point)
7. [**2-4 sentences**] Discuss what your feature extractor does, and why you believed the features would be informative. (2 points)

3 Embeddings

While word-based features are straightforward to implement, they may not capture the semantic meaning of words effectively. Remember, your classifier doesn't have any innate knowledge about what TOKENID-4 means, and has to learn a useful representation from scratch. To address this limitation, you will explore using pre-trained word embeddings. Word-embeddings map each word to a dense vector representation of fixed length; this representation can then be used to compare words (e.g. via cosine similarity) and in some cases can be used for downstream tasks like creating analogies.

You will be using the *word2vec-google-news-300* word embeddings, more information is available at the links listed here: https://huggingface.co/fse/word2vec-google-news-300

For your task, you will need a way to combine word embeddings into a fixed-length feature space to feed into your Logistic Regression classifier. One way of doing this is called *mean/average pooling*, where you get the word-embedding for each word in your text, and average them together. The idea is that by taking the average of all word embeddings you get a text-level representation that doesn't increase/decrease in size relative to text-length.

After implementing this featurizer you will try training your Logistic Regression classifier with it:

Run the training with:

```
$ python run_classifier.py --model LR --tokenizer NONE --feats WV --
    learning_rate YOUR_LR --batch_size YOUR_BATCH_SIZE --epochs
    YOUR_EPOCHS
```

And relevant unit-tests with:

```
$ python unit_tests.py MeanPoolingWordVectorFeatureExtractorTest
```

You will also use gensim to train a Word2Vec model, and compare its word embeddings against the pre-trained model via the most-similar vocab items for some pre-selected words. Run this comparison with:

```
$ python similarity.py
```

Tasks (13 points)

1. Use word embeddings for classification (**models.py**) (6 points):
   - Implement MEANPOOLINGWORDVECTORFEATUREEXTRACTOR class to create document-level features from word embeddings (via mean pooling) (4 points)
     (a) Implement GET_WORD_VECTOR() - gets word vector for a given word, if it's in the vocabulary. Return 'None' if it's not. (2 point)
     (b) Implement EXTRACT_FEATURES() - converts a list of words into a mean-pooled vector (2 points)
     (c) Note: this featurizer will use the given ReturnWordsTokenizer that splits sentences into words and punctuation
   - Train your logistic regression model using these embedding-based features (note: word-vector based feature extraction may take noticeably longer to train) (2 points).
     (a) As before, identify hyper-parameters that produce at least 80% accuracy on the test set and runs in **under 5 minutes**. Report your hyper-parameters and commands here (2 points):
2. [2-3] sentences Compare the general performance of embedding-based features to count-based features. Hypothesize as to why one performed better than the other, and what the advantages+disadvantages there are with that approach. (4 points)
3. Compare a Word2Vec model trained on the imdb dataset with the pretrained one (**similarity.py**) (3 points).
   - Implement TRAIN_WORD2VEC_MODEL() - given a list of sentences, train a basic Word2Vec model and return it (1 point)
   - [2-3] sentences We have provided some select terms and the most similar vocabulary items from 1) your Word2Vec model and 2) the pretrained *word2vec-google-news-300* model. These are extracted by getting the embedding for the given word, and finding the vocabulary items with the closest embedding. Run **python similarity.py** to see them. What trends do you see in the differences between your model and the pre-trained model? What might causes these differences? (2 points).

4 Looking Forward

Congratulations on building a sentiment classifier from scratch! It's likely you saw high performance (perhaps surprisingly high) while training your model. However, real-world applications often find that the data your models will encounter is slightly different than the nice dataset you used for training. We sometimes call this problem *distribution shift*, or say that we want a model that *generalizes* to unseen domains. By *generalize*, we usually mean that a model performs well on new data. This is part of the reason we often select our models by performance on a *dev/validation* set, as we expect performance on this non-train data to be indicative of performance on other unseen data. Note: while the obvious answer is to just get more data, it is often expensive and inefficient to collect data to cover each possible case - we would rather our models were *robust* to reasonable distribution shifts.

Some distribution shifts are subtle, for example your model have learned features specific to the movies and time periods in its dataset. As movies come out, and as the language we use to talk about movies changes, (e.g. saying an action movie is Marvel-esque), your model's performance may degrade. You may also want to use your IMDB-trained model on another movie review website, but find that the style of reviewing is so different that the features your model relied on aren't as prevalent.

Another significant challenge in natural language processing is adapting models to work with different languages. You can think of this as a kind of distribution shift, where the underlying task is the same but the way a positive or negative sentiment is expressed has been (drastically) changed to that of a different language. A model that *generalizes* well to other languages would be trained on one language, but still exhibit high performance on a different language.

Tasks (8 points)

1. [4-6 sentences] The IMDB data is entirely in English. Imagine we wanted to predict the sentiment of Japanese reviews using your trained pipeline. Remember your word-splitter, tokenizer, count-based feature extractor, and model are all 'trained' or initialized based on your training data. For each of (word-splitter, tokenizer, count-feature extractor, learned weights) describe what your model's default behaviour will be and the effect on performance. (5 points)

2. [2-3 sentences] What would you suggest to improve your model's performance on Japanese? Describe why you think these changes would improve performance. (3 points)