Advanced Natural Language Processing

Assignment 1: Twitter hate speech detection using Naive Bayes and Logistic Regression

Credits:

---

1 Introduction

In this assignment, you will implement and work with a Naive Bayes and a Logistic Regression classifier. (Note that for the Naive Bayes section, you don't need to represent the input as a vector necessarily. You can directly look at the presence of words, and look up the class conditional likelihood.)

We will use a Twitter dataset classified into "hate speech" and "non hate speech" (in our data, we have called these classes "offensive" and "nonoffensive" to avoid the charged and inaccurate term "hate speech"). For this assignment, do not use any external packages (NLTK or any others) except where specified.

2 Getting started

You should start by unpacking the archive `assignment1.zip`. This results in a directory `assignment1` with the following structure:

```
- data/
--- NAACL_SRW_2016.csv
--- NAACL_SRW_2016_tweets.json
- model/
--- __init__.py
--- logreg.py
--- naivebayes.py
- assignment1.py
- utils.py
- evaluation.py
- helper.py
```

3 Twitter Data

The available utility function will take care of the data loading and split the data to training and test sets for you. If you are interested in the datasets, you can take a look at the `data` directory. Each item in our data consists of a tuple of the tweet text and its label (represented as a string). The tweet text has been tokenized and is represented as a list of words. We can look at an example item:

```
(['At', 'this', 'rate', ',', "I'm", 'going', 'to', 'be',
'making', 'slides', 'for', 'a', 'keynote', 'in', 'my', 'car', 'as', 'I',
'drive', 'home', '.'], 'nonoffensive')
```

4 Starter Code

Let's look at the main program in `assignment1.py`. It is supposed to ...

1. reads in the Twitter hate speech dataset using `read_hate_tweets()`,
2. initializes either a `Naive Bayes` model or a `Logistic Regression` model, and
3. trains the model and then uses the model to evaluate the test data.

However, there are several missing features that need to be implemented. Let's implement them one by one!

4.1 Evaluation [15 pts]

The first thing you're being asked to do is to implement evaluation functions for a classifier and a given labelled test set in `evaluation.py`. Assume that the classifier has a `predict()` function that takes an item in the form of a provided example above and predicts a class for that item. Write evaluation functions to compute the accuracy and F1 score for such a classifier. (To test your functions without having access to a real `predict()` function, you could simulate one that makes random predictions.)

4.2 Naive Bayes Classifier [35 pts]

Next, implement the Naive Bayes classifier from scratch in `model/naivebayes.py`.

Some notes and requirements for the implementation:

- You should allow for an arbitrary number of classes (in particular, you should not hard code the two classes needed for the given dataset).
- The vocabulary of your classifier should be created dynamically from the training data. (The vocabulary is the set of all words that occur in the training data.).
- Use additive smoothing with a provided parameter k.
- You may encounter unknown words at test time. Since we're not allowed to "peek" into the test set, we will implement the following simple treatment: We will assume that we don't know anything about unknown words and that in particular, their presence does not tell us anything about which class a document should be assigned to. Therefore, we will not include them in the calculation of the (log) probabilities during prediction, under the assumption that their probability does not differ hugely between the different classes (probably not a correct assumption, but the best we can do at this point). Since we don't need correct probabilities but only most likely classes, just ignore unknown words during prediction.
- Use log probabilities in order to avoid underflow.

  Afterwards, evaluate your classifier by training and testing it on the given data. Vary the smoothing parameter k. What happens when you decrease k? Plot a graph of the accuracy and/or f-score given different values of k with `matplotlib` and save the graph(s). **Discuss your findings**. (In order to do this, use the flag `--test_smooth` when running `assignment1.py`. Implement the functionality in `train_smooth()` under `helper.py`)

4.3 Feature Engineering [20 pts]

We mentioned that the Naive Bayes classifier can be used with many different feature types. Try to improve on the basic bag of words model by changing the feature list of your model. Implement at least two variants using `features1()` and `features2()` in `model/naivebayes.py`. For each, explain your motivation for this feature set, and test the classifier with the given data. **Briefly discuss your results!** (In order to do this, use the flag `--feature_eng` when running `assignment1.py`. Implement the functionality in `train_feature_eng()` under `helper.py`)

Ideas for feature sets that were mentioned in the class include:

- removing stop words or frequent words
- stemming or lemmatizing (you can use NLTK or spacy.io for basic NLP operations on the texts)
- introducing part of speech tags as features (how?)
- bigrams

4.4 Logistic Regression Classifier [30 pts]

Implement a logistic regression classifier using the definitions given in class and gradient descent in `model/logreg.py`. For this, you will have to use a matrix representation for your data to keep track of each feature's weights per class, which you can implement using the `numpy` package.

Start by implementing a function `featurize()` that converts the (training or testing) data into a matrix format. This function should return a pair of NumPy matrices X, Y where X is an $N \times F$ matrix (N: number of data instances, F: number of features), and where Y is an $N \times 2$ matrix whose rows have one of two forms:

- [1, 0] if the gold-standard annotation class for the corresponding tweet is 'offensive', or
- [0, 1] if the gold-standard class for the corresponding document is 'nonoffensive'

This kind of representation is known as a one-hot encoding. You can read the first vector as saying that 'there is a 100% chance that the instance belongs to the "offensive" class and a 0% chance that it belongs to the "nonoffensive" class', and similarly for the second vector. Note that these are the two extreme cases for the conditional probability distribution $p(k|x)$ for class k and feature vector x.

To implement the `featurize()` function, you will need to assign to each word in the training set a unique integer index which will identify that component of the feature vector which is 1 if the corresponding word is present in the document, and 0 otherwise. This index is built by the helper function `build_w2i()`.

Your next task is to complete the implementation of the `LogReg` class. The methods `p()` and `predict()` yield the probability of a class given an item, and the best class for the item, respectively. They can be implemented using appropriate NumPy matrix operations and the provided `softmax()` function. Note that you should set up both methods to take a whole matrix of input vectors as input, not just a single vector.

The training procedure is implemented in the (class) method `train()`, using iterative optimization. Typically, we shuffle the training data and split them into mini-batches (e.g, 100 items), then update the weights after each minibatch. This is done for `max_iter` number of iterations, or "epochs". Each epoch iterates over the training data set once.

Lastly, implement the missing methods using L_2 regularization with parameter $C = 0.1$ and train the Logistic Regression model for 10 epochs with the default learning rate η. Define the complete training and testing functionality in `train_logreg()` under `helper.py`

5 Submission

Upload your `assignment1_LASTNAME.zip` file on Moodle.