Introduction

In this lab you will explore sentiment analysis using tweets from Twitter. This data is taken from SemEval-2015, Task 10. You can find more information about this task here: http://alt.qcri.org/semeval2015/task10/

SemEval-2015 is an international semantic evaluation workshop that is taking place next year. The workshop invites participants from around the world to submit their best solution to open problems in computational semantics. In order to best compare how each of the participant’s systems perform, each group uses a common training and test set. The test data is available now, but the training data will not be released until December 5.

For this lab, we will focus on Task 10, subtask B: “Given a message, classify whether the message is of positive, negative, or neutral sentiment. For messages conveying both a positive and negative sentiment, whichever is the stronger sentiment should be chosen.”

Examining the data

The data provided by the Semeval-2015 competition is in /data/cs65/semeval-2015/B/. There is no test data yet – only training data. Like last week, before you begin work on the lab, you should to spend a few minutes looking at the source data. The file you want to look at is twitter-train-full-B.tsv. Each tweet is classified with one of the following labels: negative, positive, neutral, objective, and objective-OR-neutral.

Parsing the data

The source data for the Senseval task is not particularly tricky to parse correctly, but I have provided you with a parser that puts the data into the same format that you used last week. You can find that in parsetweet.py. Unlike last week, there are not separate lexelts – it’s just a bunch of tweets that you want to classify. To make things easier, though, the data format is the same with a single lexelt called ‘tweet’.

""
Contents of warmup.py
""

from parseTweet import parse_tweets

if __name__=='__main__':
    filename = '/data/cs65/semeval-2015/B/train/twitter-train-full-B.tsv'
    tweetData = parse_tweets(filename, 'B')

Like last week, you’re also welcome to modify the parseTweet.py file, though it is not necessary. Here are some examples using the tweet data:

$ python3 -i warmup.py
>>> tweetData.keys()
dict_keys(['tweets'])

$ python3 -i warmup.py
>>> tweetData['tweets'].keys()
# these will show up in a different order each time you run it
dict_keys(['260149911914414080_43952467', '237726495244685313_228203206', ...])

```python
>>> tweetData['tweets'][237726495244685313_228203206]
{'words': ['Lakers', 'vs', 'Heat', 'on', 'Jan.', '17th!', 'It\'s', 'D', 'Wade\'s', 'b', 'day...', 'I', 'feel', 'bad', 'he\'ll', 'lose', 'on', 'his', 'birthday', 'lololol'],
'answers': ['negative'], 'heads': []}
```

Notice that there are no ‘heads’ in this data. You are not analyzing the sense of a particular word, you are classifying the sentiment of the entire tweet. No word is a “head”. You can still use your decision list code, but your feature extraction is going to have to change.

Exploring the training data

You should store the solutions to questions 1-4 of the lab in `warmup.py`.

1. How many training examples are there?

2. How often (in raw numbers and percentages) do each of the sentiment labels show up? Report the numbers with the sentiment labels as given. Next, report the numbers a second time but this time you should conflate **neutral**, **objective** and **objective-OR-neutral** into a **neutral**. (You’ll want a way to turn this conflating on or off, so try to make this part modular.)

3. What is the random baseline?

4. What is the “most frequent sentiment” baseline? Find the most frequent sentiment for the tweets and the label every training instance with that most frequent sentiment. What percentage of the instances in the training data would you label correctly using the most frequent sentiment baseline? Repeat by conflating the neutral/objective labels as you did in Question 2.

Cross-validation

Try to make the code you write from here to be modular so you can use the code for other things. Ideally, you’d keep the decision list classifier in `declist.py` and the naive bayes classifier in `naivebayes.py`. You could even put the cross-validation code into `crossval.py`. Then, import each of those into a `tweet.py` file which will call either the decision list or the naive bayes classifier depending on what you’re trying to do, all while using cross-validation. Please put your text answers to all of the remaining questions in `tweets.txt` or `tweets.tex`.

On the previous lab, you repeated many of the experiments above on the test data. On this lab, you do not have test data. But, you can make some! The idea of cross-validation is that you hold out some of the training data and use it as test data. Then you repeat the experiment again, holding out a different portion of the training data. You repeat this until you’ve eventually held out each section of the training data exactly once. Typically, we would divide the training data into 5 equally sized pieces (5-fold cross-validation) or 10 equally sized pieces (10-fold cross-validation). For example, in 5-fold cross-validation you would break your training into 5 chunks, which I’ll call A, B, C, D, and E. In the first pass, you’d train on A, B, C, and D, then test your system on E, counting how many examples you got correct. Then, on the second pass, you’d train on A, B, C, and E, and test your system on D. You’d repeat this until you’d used each chunk as your test data once. Typically you would randomize which data ends up in each chunk, but don’t do that yet as it will make testing your code much more difficult.

For the remaining questions in this section, we will use 5-fold cross-validation. Make 5 a parameter somewhere so that you can change later to 10 or 20 or whatever else you want! In the discussion below I
will talk about training data and test data. However, what I mean when I say that is that you are
really using 80% of your training data as the training data and 20% of your training data as test data,
and that you are repeating the experiment 5 times (since you are doing 5-fold cross-validation).

For each of the questions below, conflate the neutral sentiments into a single sentiment as you did in
Question 2.

5. For each chunk of your cross-validation, find the most frequent sentiment. Is it always the same?
   Is that surprising?

6. Use the MFS from the “training data” to label the “test data”. How well does this do?
   (Remember: you are doing cross-validation here.) How does this match your intuition? Discuss.

7. Use bag of words features and your decision list classifier from last week to classify each of the
tweets. Does your decision list outperform the MFS baseline?
   - Do stopwords help?
   - Does case-folding help?

8. An important item that we’ve omitted so far is that some tweets contain words that are
   associated with positive sentiment (e.g. “good”) but are actually negative because they’re
   preceded by a negation (e.g. “not good”). One way you can handle this is to change every word
   in the tweet after the word “not” and before the next punctuation mark so that it is negated. For
   example, you could change “this is not a very good example, but that one was.” to be “this is not
   NOT_a NOT_good NOT_example, but that one was.”

Writing a naive bayes classifier

Ideally, if we’re given a feature vector \( \vec{f} \), we’d like to choose the best sentiment label \( \hat{s} \) out of the set of
possible labels \( S \) such that its probability is maximized:

\[
\hat{s} = \arg\max_{s \in S} P(s|\vec{f})
\]

Using Bayes’ rule, we can rewrite the formula as follows:

\[
\hat{s} = \arg\max_{s \in S} \frac{P(\vec{f}|s)P(s)}{P(\vec{f})}
\]

We can omit the denominator \( P(\vec{f}) \) because it is constant for all sentiment labels. If we then make the
naive assumption that each of the features in \( \vec{f} \) are conditionally independent given the
sentiment label, we can rewrite the formula as follows:

\[
P(\vec{f}|s) \equiv \prod_{j=1}^{n} P(f_j|s)
\]

Given the above rewrite, we have the following naïve bayes classifier formulation:

\[
\hat{s} = \arg\max_{s \in S} P(s) \prod_{j=1}^{n} P(f_j|s)
\]

We can estimate \( P(s) \) using the maximum likelihood estimate from the training data:
\[ P(s) = \frac{\text{number of training instances labeled with sense } s}{\text{number of training instances}} \]

We can also estimate each of the individual feature probabilities using MLE:

\[ P(f_j | s) = \frac{\text{count}(f_j, s)}{\sum_i \text{count}(f_i, s)} \]

Even now, some of the \( P(f_j | s) \) values are likely to be zero, so we’ll need to do some kind of smoothing to prevent multiplying by zero.

9. Implement the naive bayes classifier described above. Use 5-fold cross-validation to train and test your classifier. Does your naive bayes classifier outperform the MFS baseline and/or the decision list classifier?

10. As before, try your classifier with and without: stopwords, case-folding, and negation. Try each separately and try some promising combinations.

11. We might want to consider tokenizing the data so that tweets that contain text such as “tonight.....the” are treated as “tonight ..... the”. This is a tricky task because you have to be careful not to mess up hashtags (e.g. “#MNFootNg”) and handles (e.g. “HeatherNoel13”). Fortunately, there is an open-source tool that does tokenization that you can use. See http://www.ark.cs.cmu.edu/TweetNLP/. I put the code and a small Python wrapper in /data/cs65/semeval-2015/arktweet. I wrote the Python wrapper but the tokenizer was not written by me. If you have problems with the code, though, that’s likely my fault, so let me know! Sample usage is in the arktweet.py file.

Once you’ve got tokenization working (it shouldn’t take you more than a few minutes to do so), re-run your decision list and re-run your naive bayes classifier. Does tokenizing improve performance?