Carlos Bros

Spam Filter Improvement Report

- ## General Design of the System

  This spam filter is designed around a SpamFilter Python class which has a constructor that takes in two strings representing system paths. One is the path of a training set of spam emails, and the other is a training set of non-spam emails. The SpamFilter trains itself using these two sets of emails to then, given an email (is_spam method), determine whether such email is spam or not.
  The design of my system is based on the Naïve Bayes Model and Bayes Theorem, building a generative model that approximates how data is produced. By using prior probability of each category (training), it can produce a posterior probability distribution over the possible categories (spam or not spam), given a description of an email. If the probability of spam is greater than the probability of not spam, then the model concludes that the given email is spam.

- ## Features

  - Unigram tokens (defined as any string or word surrounded by spaces).
  - Bigram tokens (defined as any two unigrams combined, separated by spaces).
  - Length of unigrams (separated into unigrams of length less than 7, of length between 7 and 13, and of length greater than 13).
  - Numbers (i.e. taking into account whether tokens are numbers or not).

- ## Preprocessing and Tokenization

  The preprocessing stage is the training stage of the spam filter.
  When the application receives the two directories containing training sets of spam and non-spam emails, it creates two different dictionaries, one for the spam category, and one for the non-spam category.
  These dictionaries have tokens as keys (both unigram and bigram, as explained above) that map to the log probability of a token occurring given a category (spam or not spam). This is calculated as follows, for a token *w*, a category *c* (spam or not spam), smoothing constant α and $|V|$, the number of tokens:

$$\hat{P}(w \mid c) = \frac{\text{count}_c(w) + \alpha}{\left(\sum_{w' \in V} \text{count}_c(w')\right) + \alpha(|V| + 1)}$$

  The reason we take the log of these probabilities is because we will be multiplying all of them together, and so multiplying numbers between 0 and 1 can result in decimal underflow. Thus, instead we log the probabilities and then later add all the logs for each category (because adding all the probability logs is the same, by definition of logarithm, as taking the log of the multiplication of the probabilities).

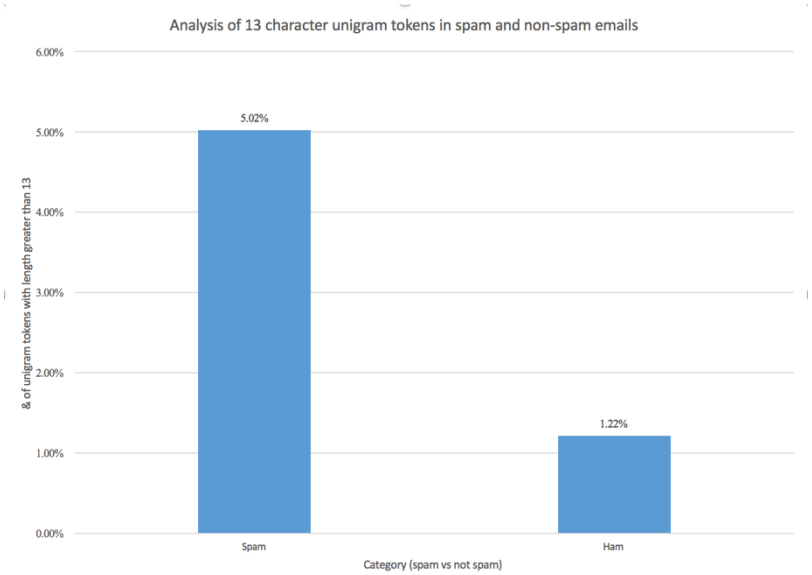
The probabilities explained before are done for unigrams and bigrams (I used different smoothing constants for unigrams and bigrams, and chose the one that incremented the accuracy on the development set the most - 1e-9 for unigrams and 1e-15 for bigrams). However, when we process a new email for which we have no information, we might encounter tokens for which we have no entry in the preprocessing dictionary of log probabilities. For that case alone, during the preprocessing we create a dictionary entry with the key "<UNK>" with the same formula, except the count(w) is now 0.
Regarding the feature related to the length of unigrams, I created my own keys that represent individual bins in the log probabilities dictionary: "<< len<7 >>", "<< 7<len<13 >>" and "<< len>13 >>". In each of these bins will go the log probability as calculated for any token, except the count(w) in the numerator will now be the number of unigrams satisfying the respective length (less than 7 characters, between 7 and 13, and more than 17 characters, one for each bin). The same applies for tokens that are entirely numbers (with a separate bin with key "<< digit_count >>").

- ## Experiments

My main experiment to test the system was directly calculating the accuracy percentage on the provided development set when training on the provided training set. The way I did it was by running the is_spam method on every spam file, and dividing the number of flagged emails over the total number of emails. A similar calculation was done for the non-spam emails.
The original spam filter (last week's homework) was 97% accurate. After adding the bigram feature, the accuracy went up to 97.75%. Once I added the length and digit features, I was able to bring the accuracy up to 99.5% (code that tests it is commented out at the end of my code).

- Length of unigram tokens:

The percentage of words with more than 13 characters is significantly different in spam emails vs non-spam emails, as seen in the following chart:



- Choosing the best possible smoothing constant:
I used different smoothing constants for unigram and bigram tokens. I basically tried different smoothing constants with the goal of maximizing accuracy, and found that the bigram smoothing constant generally will work better if it is smaller than the unigram smoothing constant. This makes sense, since there are many more bigram tokens and so applying a larger smoothing constant could skew our results. In order to further minimize how skewed the results are, I also eliminated any bigram tokens that only appeared once.

- Error analysis

  In the given development set, my spam filter fails to recognize as spam file data/dev/spam/dev283. The email is really well written and talks really formally about a serious topic. Thus, it uses words that are more typical of a non-spam email, and that is most likely why it was flagged as not spam.
  The only other file my spam filter fails to recognize is a non-spam file (which my filter flags as spam). This file is data/dev/ham/dev118. This non-spam email is very short and happens to contain many long tokens, so it is most likely that the length of token feature was the cause of this email being flagged as spam.