MaxEnt Classifier

Bruce Spang

December 3, 2013

`https://github.com/brucespang/maxent`

1 Abstract

For this project, I built a fast maxent classifier to guess the gender of people by their first name. While the task is seemingly uninteresting on its own, it provides a good way to test a maxent classifier. Additionally, it is a useful feature for in-document coreference resolution and the accuracy of my features reveal some interesting things about people's names.

2 Building a fast maxent classifier

I initially implemented the maxent classifier in very idiomatic python using the SciPy optimizer. It worked, but was way too slow: it took 1 minute to train 10 features on 50 names for 2 classes.

C is faster than python, so I rewrote the core training routines of the classifier in C, using GSL for the numerical optimization. I implemented it as a python library, so that I could still use features written in python (since it seemed easier to do string manipulation in python). It took a few seconds to do the test that took the python version a minute, but when I scaled up and trained on 30 features and 500 names, it took a few minutes.

I realized that the C version was jumping into python whenever a feature was evaluated. The implementation was also re-evaluating every feature for every calculation on every iteration of the training algorithm, which turned out to be 45 million times. Since the features I wrote produced the same output given the same input, I memoized them. After adding the hashtable from Concurrency Kit, things were much faster. I was able to train on 30 features and 8000 names in a few seconds. Unfortunately, I figured it'd be a good idea to add features for the position of every character in the name (300 features in total). Even with just 500 names, training too long. I killed the process after five or six hours, but it could have easily gone on longer.

I decided that I really wanted those extra features, so I did some research into properly implementing MaxEnt parsers. I found a paper (http://www.cs.grinnell.edu/ weinman/pubs/weinman11large.pdf) that described implementing MaxEnt with matrix multiplication. I rewrote my implemention using GSL's matrix facilities based on the matlab code in the paper. There were a few complications:

- The paper doesn't mention how to calculate the empirical count with matrix operations. I wound up writing something using for loops that accomplished the same thing, but was probably slower.
- GSL doesn't support using matricies as input to the function optimizer. I had to treat the underlying chunk of memory for the matrix as a vector in order to get the optimizer to work.

Once I had the matrix implementation working, things were much nicer. That same test that took over six hours with the previous version only took a second. I added a whole bunch more features (e.g. the presence of all combinations of letters), and more data. With 1000 features and 10k training examples, the matrix implementation only took 40 seconds to train. There is definitely more work I could do to make ththis implementation more efficient (e.g. using more vector operations, thinking more carefully about how the hardware executes the algorithm, being more careful with how I allocate memory, rewriting the features in C, etc...), but I'm happy enough with its speed.

As a side note, the paper on the matrix implementation has a different definition of features. Instead of being a function of a class and a data point, they are only a function of a data point. Instead of being vectors, this means that weights are a matrix of features and classes. However, I found myself never using the class when inventing feature (e.g. I thought of features like "the first letter of the name is a vowel", not "the gender is female and the first letter is a vowel"). Taking the cross-product of each class with the features seems equivalent to using a matrix as in the paper.

3 Features

I used the following 1220 features to classify gender by name:

- name length
- number of vowels in the name
- number of consonants in the name
- ratio of vowels to consonants in the name
- the presence of a letter in the name
- the presence of a pair of letters in the name

- for each of the first 16 positions, the letter at that position
- for each of the first 16 positions, whether the letter at that position was a vowel
- for each of the last four positions, the letter at that position
- for each of the last four positions, whether the letter at that position was a vowel

4 Evaluation

I trained the classifier on approximately 8000 names from item of male and female names included in NLTK. I picked 500 other names from NLTK to use for evaluation. I randomly sorted the names, so that I wasn't e.g. training on male names and testing on female names. I measured the percentage of names classified correctly.

With the python implementation and the first four features (and whether the first two or last two letters were vowels), the name-gender classifier was about 60% accurate. Unfortunately, this does worse than the baseline of just picked "female" every time, which is about 65% accurate.

With the matrix implementation and the features from the previous section, the classifier is about 85% accurate.

It is surprising that letter positions and whether or not letter are vowels is able to classify gender with such high accuracy. I would have expected it to need to memorize items of male and female names. This suggests that people might rely on things like the sounds of names to determine the gender of a name, instead of something like frequency of people that we personally know with the name.

5 Implementation Notes

Numerical algorithms are horrible to debug. Once they compile, they run for awhile and then print out some output. There's no easy way to tell if the output is correct. I only tended to notice bugs when they consistenly classified everything one way or another, or gave every feature a weight of zero. Fortunately, MaxEnt classifiers are robust to programming errors.

Once I realized there was a problem, I then had to figure out where things were going wrong. This was usually non-obvious: because the optimizer usually just stopped after one iteration and gave no indication as to why. The following are some bugs I encountered:

- I passed the wrong variable to a function, which resulted in the function value being independent of the training data, and the optimizer terminating early.

- I returned the gradient vector instead of the weights vector after training. Because of conversion between C doubles and python floats, the weights were on the order of 10^{-19} instead of 0. This meant that it would decide that everything was one class or another. Since the training data is about 35% male and 65% female, this led to occasional 30% jumps in accuracy when I added a random feature.

- I used an unsigned integer instead of a double at one point when adding up weights in a column to normalize. Since the weights could be negative, this occasionally caused integer underflow which made the normalized results meaningless.