String Regularization

Tamara Andrade

December 13, 2013

1 Introduction

Patterns in textual data are easy for humans to spot. Humans are good at ignoring small permutations to focus on larger similarities. For example, the phrases "Example 1" and "exampl e 1" are easily recognized as being approximately the same by humans. However, computers struggle to see these larger patterns. "Example 1" and "exampl e 1" read quite differently to a computer, just as "Example 1" and "Example 2" would.

This is a big problem when it comes to analyzing databases. A researcher would want the computer to know "Example 1" and "exampl e 1" are the same entity, but the computer treats them as distinct. Researchers have two choices. Either they can ignore these problems and essentially throw out some information, or, if their data is not large enough to safely discard this information, they can manually clean the data. A lot of researchers spend a lot of time and money cleaning (or hiring someone to clean) their data.

My project aims to help researchers spend more time analyzing their data and less time cleaning it. To this end, it aims to automatically identify typos and suggest corrections when it has been given a list of strings. This project is aimed at medium sized databases: those that are too large to easily clean by hand but too small to analyze properly without cleaning.

2 Choosing the algorithm

At first brush, this would seem like an unsupervised learning problem because there is no training data. In that case, the problem is one of clustering: the goal is to group words together and then extract the right meaning. If this were the case, k-means and expectation maximization (EM) would be the right algorithms to choose.

However, these approaches have a major flaw for this application. While clustering is easy based on numerical value (1 and 3 are both closer to 2 than they are to 15), the approach is much slower when it comes to text. Measures of textual distance like Levenshtein distance[1] exist, but they are valid largely only when comparing two strings. This means that clustering strings requires an enormous number of calculations between all the strings in the database.

I chose to go with a different approach. Instead of using an unsupervised algorithm, I tweaked an algorithm from supervised learning – Naive Bayes – so that it would work in an unsupervised setting. The main advantage to this approach was processing time. Naive Bayes still requires the comparison of a large number of strings, but it can be adjusted to make fewer comparisons without throwing out any relevant information.

Naive Bayes as a choice makes sense if you think of the problem as not one of clustering, but fundamentally one of spelling correction. Most commercial spell correctors use some kind of Naive Bayes. They solve the following problem:

$$pr(\text{ candidate word } | \text{ what typed }) = \text{argmax}\Big(pr(\text{ what typed } | \text{candidate word }) * pr(\text{ candidate word })\Big)$$

[1] Levenshtein distance is a count of the number of transformations – either addition, subtraction, or transposition – needed to change one word into another. This knowledge comes from CS124.

The difference is that most spellcheckers have to be trained on a large collection of text, a corpora. They derive some model of English (from which they know which words are more likely to appear) and a model of common typos (from which they get the probability that you would have typed what you did given a candidate word.[2]

3 My algorithm

I apply Naive Bayes to an unsupervised setting by developing a model of these probabilities based on the data. I assigned the probabilities as follows:

$$pr(\text{ candidate word }) = \frac{\text{count of times word observed}}{\text{size of the data set}}$$

$$pr(\text{ what typed } | \text{ candidate word }) = \frac{1}{1 + \text{LevenshteinDistance}(\text{ what typed , candidate word })}$$

The chief advantage to this approach is that the comparison in terms of Levenshtein Distance only needs to be done between unique words. No information is being lost because word counts is being stored separately and entering into the equation through the probability put on different candidate words.[3] In contrast, you would not want to take the unique instances of the data in clustering because it would increase error. The difference in runtime on my machine is startling. On one dataset, k-means was taking 30 minutes to run. Implementing this version of Naive Bayes reduced it to less than 3 minutes.

There are some weaknesses to this approach. One might think there are some times when word count will be less informative. For example, if the data was naturally skewed so that the "true" observations appeared at different rates, you should rely less on word counts. For this reason, I added a weight to these two probabilities (set by the user)[4] so that they can decide for themselves how much they want to rely on one kind of probability versus another.

Another way to think through this would be to add some kind of Laplace smoothing parameter. Initially, I tested a few instances of Laplace smoothing and found that weights did a better job on my two datasets. However, with more time, I will be testing these different parameters more rigorously on a wider variety of datasets than I was able to do during this quarter.

The other design choice was how to penalize edit distance. Conventional spellcheckers use information about common typos. For example, replacing "m" with "n" would intuitively seem to occur more often than replacing "m" with "q". This occurs both because "n" is closer to "m" on the keyboard, looks more alike, and makes a sound closer to "m" than "q." However, I was reluctant to take this approach. I wanted to be more agnostic about the sources of errors. Most common spellcheckers encounter errors of only one type – people mean to type one thing and they type another. That same error generating process might not be true for many datasets, however. For example, someone could alternate between writing "Mr. Fox" and "Mister Fox". This is "wrong" in the sense that the machine cannot recognize these two entities as the same, but it does not follow the same pattern as typos. I therefore settled upon the above formula as something that would penalize a smaller number of mistakes more, but then taper off as the number got larger (the difference between 5 and 6 typos is not big, but the difference between 0 and 1 is.)

In deriving this probability, I thought about it in terms of a penalty function – a function that would assign less probability to candidate words that required more edits to get to the word. However, as was pointed out to me in the poster presentation, this is not actually a true probability in that the total does not sum to one. In the future, I will think about different ways to conceptualize this variable and see the effect it has on performance.

4 The data

I wanted to ensure that my algorithm was checked on data that had very different distributions in terms of the true number of "types" – to see how the data performed there – as well as a number of different error generating processes.

[2] Once again, thanks to CS124 for this information about spellcheckers.

[3] These candidate words are all unique words in the dataset.

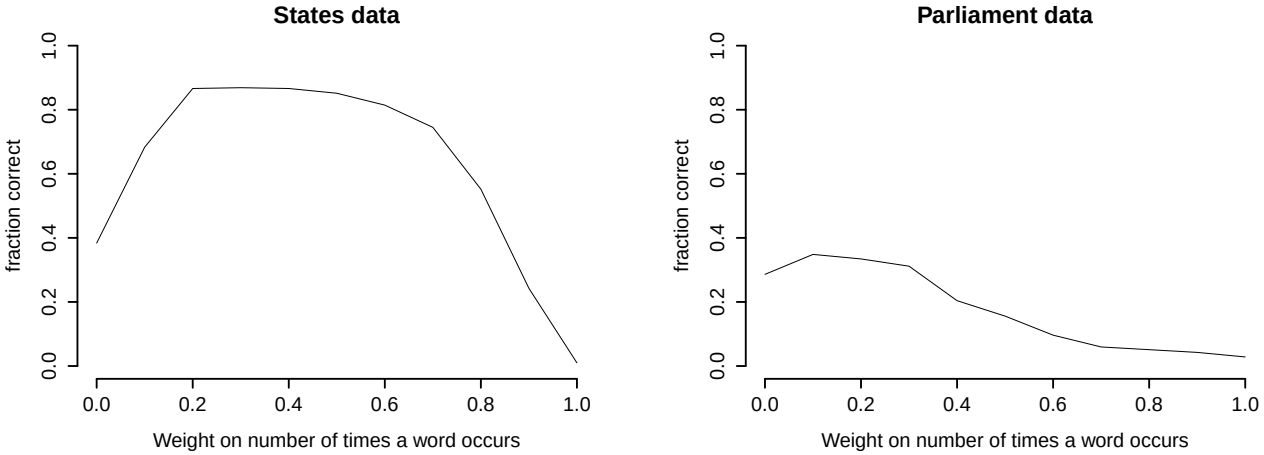
[4] The weights are complementary: the user sets a number between 0 and 1 and this is subtracted from 1 to get the other weight.

I created data sets to test both of these kinds of models. The first was a list of the states in the United States, each typed 50 times. Typos were not corrected. Therefore, odd spellings like "alalmba" are present in the data. This dataset allows me to see how the algorithm works when the errors are at random and when the true distribution of types is roughly equivalent.

The second was a dataset generated from historical transcripts of the British Parliament. These records were taken from online records, and the errors represent the inconsistent spellings present in any historical records. "Chancellor of the Exchequer" gets spelled "Chanceller of the Exchequer", "Chancellor of Exchequer", etc. Since these data were based on the number of times individuals spoke in Parliament, the distribution is highly skewed. The Chancellor of the Exchequer speaks a lot; some quieter members appear only once or twice in the dataset.

5 Performance

It is not exactly surprising, given the more challenging nature of the Parliamentary data, that it was easier to generate higher correction rates for the state data.



It was reassuring, however, to see both very high rates of success in the states data and to still see some gains in performance using this approach even on the challenging data.

I was expecting the alpha to skew closer to 0.5 for optimizing performance on the state data, but it was actually significantly to the left. This would seem to imply that a better model is needed, as the best performance still needs to rely more on one part of the probability.

6 Limits of this Project

When designing this approach, I was cognizant of some limitations. First, this algorithm can only generate corrections that are actually present in the data. For example, the user might desire "Mr. Fox," to be changed to "Mr. Fox". However, if the only options in the data are "Mr. Fox." or "Mr. Fox,", the algorithm should standardize both cases to whichever occurs more often.

The other major limitation is that the program will not work if there are no correct spellings and every misspelling occurs at the same rate. This, however, seems like an acceptable problem, since this is clearly a case where some outside knowledge of the data would be needed regardless. For example, if someone had to choose the correct spelling and only saw "Shanghay", "Shunghai", and "Shenghai" one time each, the "best" value could only be chosen by someone with knowledge of major cities in China.

7 Future work

There are a few important areas to work on in future work. I will focus on three:

- **Testing on a wider variety of data.** I spent most of the time this quarter collecting and cleaning the data and working through which algorithm to select. The result was that I did not get to test on as much as data as I would have liked. One way to see how robust these specifications I have chosen are is to bring in a wider variety of data.
- **Make changes to improve accuracy.** Once I have a wider variety of data, I can begin to analyze different ways to improve the accuracy. For example, I can get a better sense of how much Laplace smoothing will help me in different contexts.
- **Make a user interface for this data.** After I have a sense for how well the algorithm works, I want to distribute this to a larger public. I think it has a lot of applications in my field of interest, political science, where people have a lot of messy data and sometimes lack the programming skills to clean their data in the most efficient way possible. Therefore, I want to build a website where data can be uploaded, analyzed, and then the guesses will be returned for the user to approve or decline. This will result in helping avoid errors where the machine makes a wrong guess. I am envisioning an interface where the machine gives a summary of its suggested changes, ranked from least sure to most sure. The user would accept changes up to a point where the algorithm has done so well that she is confident in the remaining changes.