PCA “Neural” Classifier

Linas Vepstas

June 6, 2017

Abstract

This describes the classifier algorithm I plan to implement, to plce words into grammatical categories. Its simple, stright-forward, and a real CPU-cycle burner. This is an algo I invented out of thin air, it resembles PCA/Markov in the early stages, a neural net in the middle stages, and true dimensional reduction in the last few steps. This sounds fancy, but really, its really very simple.

Introduction

The next step in the language-learning process is what I’ve been calling “clustering”. It really needs to be something more like factor analysis, or better yet, sparse PCA. Except that’s not right, either.

What is needed is a recognizer,as follows. Consider $\overrightarrow{b} = \sum_n b_n w_n$ be a vector, with the w_n being individual words, and the b_n being weights. Plain-old Principal Component Analysis (PCA) computes real-valued weights b_n. It’s problematic, because potentially all of the weights are non-ero for all of the words. Sparse PCA computes real-valued weights b_n such that only some small number of them are non-zero. This is much better. But what is really needed is a classifier: a set of b_n that are either zero or one, indicating the membership of a word w_n in some class of words. (Note, by the way, that a word might belong to multiple classes, for example, according to its part-of-speech, or it’s meaning.)

This suggests the following neural-netish variant on iterative PCA (entirely of my own design, cribbed from nowhere at all, just popped into my head as I sit still immobilized.)

1. Start with $b_n = 1/\sqrt{|w|}$ where $|w|$ is the number of unique words. This starting point is a unit-length vector, i.e. $\left|\overrightarrow{b}\right| = 1$. Its convenient to change notation, here, and write $b(w)$ for the value of $\overrightarrow{b}$ at word w. That is, $b(w_n) = b_n$ is the same thing.

2. Let $p(w,d)$ be the frequency matrix, as defined before: $p(w,d) = N(w,d)/N(*,*)$, and where $N(w,d)$ is the number of times word w has been observed with disjunct d. As noted earlier, $N(w,d)$ is very large and very sparse: typically $200K \times 4M$ in recent datasets, with only 1 entry out of 2^{15} being non-zero.[1] Compute the

[1] I plan to send out the revised, expanded statistical analysis “real soon now”.

double-sum

$$s(v) = \left[PP^T b\right](v) = \sum_d p(v,d) \sum_w p(w,d) b(w)$$

which is basically a pair of dot products. Its still a large, time-consuming computation, even for sparse vectors.

3. Normalize: set $\overrightarrow{b} \leftarrow \overrightarrow{s} / \left|\overrightarrow{s}\right|$ so that $\overrightarrow{b}$ is of unit length.

4. Repeat these steps k times: go to step 2 and run the summation again. The repetition here is the 'power iteration' or the 'von Mises iteration' method for computing the largest eigenvalue of $\left[PP^T\right]$. It is not guaranteed to converge, and if it does, it might not do so quickly. But we deal with this in the next step, so its sufficient to keep k small, just enough to get a trend going. Another way to think of this is as a Markov process (specifically, a Markov chain). That is, the matrix $\left[PP^T\right]$ will behave essentially as a Markov chain, and iteration on it just identifies the primary Perron-Frobenius stable state (step 3 makes it Markovian, by preserving to total probability measure). That is, $\left[PP^T\right]$ defines a weighted adjeaceny matrix for a graph, and iteration creates a measure-preserving process (walk) on this graph.

5. After the above repetitions, apply some standard neural-net sigmoid function to $\overrightarrow{b}$. That is, set $b(w) \leftarrow \sigma(b(w))$ for some sigmoid. This has the effect of driving some of the elements to zero, and others to one.

6. Repeat this m times: go to step 2, and repeat steps 2-5. Viewing this as a dynamical system, the effect of the sigmoid function is to force the system into a block-diagonal form, with the vector $\overrightarrow{b}$ identifying a highly-connected block. Another way to look at this is as a graph factorization algorithm: the vector $\overrightarrow{b}$ is identifiying a well-connected subgraph, which is only weakly connected to the rest of the graph. The vector (viewed as a measure-preserving dynamical system) is spending most of its time in one particular block. Again, $\left[PP^T\right]^k$, the k-th power iterated matrix from step 4, can be thought of as a surrogate for a weighted graph adjacency matrix. A third way of thinking of this is as an m-layer neural net, with the link weights between one layer and the next being given by $\left[PP^T\right]^k$. All three ways of looking at this are essentially equivalent: a measure-preserving dynamical system, a chatoic and mixing process on a graph, or as an m-layer neural net. Pick your favorite.

7. Classify. Pass the vector $\overrightarrow{b}$ through the step function, i.e. $b(w) \leftarrow \Theta(b(w))$ where $\Theta(x) = 0$ if $x < 1/2$ and $\Theta(x) = 1$ if $x > 1/2$. The step function is a super-sharep sigmoid. This step identifies and isolates an active, well-connected subgraph of $\left[PP^T\right]$. It identifes a square block, of dimension $|b| \times |b|$ where $|b|$ is the total number of non-zero entries in this final $\overrightarrow{b}$. To belabor the point: the block-matrix is explicitly

$$B(v,w) = b(v)b(w) \sum_d p(v,d) p(w,d)$$

The non-zero elements of this final $\overrightarrow{b}$ identify a class of words that can be considered to be grammatically similar or identical. This is the "clustering" step.

8. Associated with this class of words is a disjunct set, the "average disjunct" for the class. It can be taken to be the set $\{d|0 < \sum_w b(w)N(w,d)\}$. The observed counts associated with this set can be taken to be $N(b,d) = \sum_w b(w)N(w,d)$ and the frequencies similarly: $p(b,d) = \sum_w b(w)p(w,d)$. From here-on, the set of words $b \equiv \{w|0 \neq b(w)\}$ can be treated as if it was an ordinary word, behaving like any other, with the indicated disjuncts, counts and frequencies.

9. Since words can have have multiple meanings, or rather, multiple different kinds of grammatical behaviors based on thier part of speech, the identified words need to be subtracted, en block, from the matrix $p(w,d)$, and then the process repeated, to identify another class of words. Put another way, if b is to be added to the set of words, as "just another word", then the frequencies $p(b,d)$ have to be subtracted from the matrix P, and shunted to this new "word", so as not to loose the overall normalization. That is, one must preserve the identity $\sum_{w,d} p(w,d) = 1$. So define, in the next iteration

$$p(w,d) \leftarrow \begin{cases} p(b,d) & \text{if } w = b \\ p(w,d) - b(w)p(b,d) & \text{otherwise} \end{cases}$$

(Hmmm. This may not be right, its late and I'm tired). This still sums to the identity except that now some of the values might go negative, and we don't want that.

10. And so we get to what should be called step zero: We want to truncate, and discard the negative entries. This should have been carried out as an actual step 0: a pre-conditioning of the matrix: some noise filtering, e.g. discarding all words that were observed less than a handful of times, discading rare or preposterious disjuncts. Pre-conditioning in this way will have the effect of removing some (possibly many) of the words from the matrix: the size of the matrix shrinks. This is the step where the actual dimensional reduction takes place: the size of the set of words is shrinking, as they get classified into sets.

11. Go to step 0 and repeat, until the preconditionaing and noise-removal has left behind an empty matrix (or alternately, a matrix where all words have been classified into some group). So, for example, words which have only one part-of-speech or meaning would (hopefully should) get classified after just one step; words that are more complex, and have two parts of speech, would require at least two iterations. This is perhaps optimistic; I expect dozens of iterations to get anything vaguely accurate.

12. There's one more step. After the formation of the class b, we arrive at a situation where no (pseudo-)connectors connect to b directly. Instead, all disjuncts connect to words inside of b. But this is a problem: we don't know if any given connector actually connects to some $w \in b$ or if it connects to the same w, but outside of b. (e.g. if b are nouns, then does "saw+" connect to "saw" the noun,

or “saw” the verb?) Thus, after some small number of iterations of step 11, there needs to be a re-parse of the entire text, using these newly discovered classes of words.

That’s it. I think this should work fairly well. Clearly, there are many nested loops, and so this is potentially a very time-consuming computation. The number of iterations k and m need to be kept small, and the classification in step 11 needs to be kept greedy, because step 12 is expensive. An alternate strategy is to brutally precondition $p(w,d)$ to make it as small as possible; but this risks throwing out the baby with the bathwater: early on, we want to cluster together the rare, obscure, unused words as best as possible into arge bins, and then devote large CPU resources to correctly classifying the remaining much smaller set of verbs and prepositions, which we know, *a priori,* to be complex and difficult, due to thier grammatical variability.

The End

References