Project in Probabilistic Models

Juho Kallio

May 11, 2016

Abstract

The algorithm to be created was a predictive system for WiFi source strength sensors. After trial and error, simple things seemed to work best: full probabilistic model was too slow, assumptions about WiFi source strength to appear around normally distributed useless. I ended up with a model that is in some sense similar to HMM. The model has the WiFi sensors present as discrete distributions of WiFi source strength frequencies, figures out likelihoods of each sensor and then the likelihoods of each strength value. This contains however the invalid assumption that sensors are independent from each others. The best score was -14883, significantly better than the required minimum score of -22700.

1 The Problem to Solve

The problem setup was to predict WiFi sensor readings based on earlier observations. We were told there was something like 100 sensors that measured strengths of 303 different WiFi sources. Sometimes the sources could move, disappear or give anomalies - it was real world data. The training data set was a 60Mb CSV file, a little below 100 000 individual observations. In the testing phase, an observation was to be read value by value, and our probabilistic model should always give the best possible guess of the discrete distribution (0-99) of the next value. The training data would contain 300 observations, which would make it 300*303*100, i.e. nine million predicted values. Not quite big data, but no room to do too much processing.

2 Implementation

2.1 Technology Choices

I wanted originally to try Figaro, a probabilistic programming library, and that dragged me to Scala. I attempted to make this one well and in the beginning I wrote quite a lot of tests. At some point though, as I ended up changing ideas a bit and gained more understanding, many of those earlier tests were no use. As usually, at some point time started to run out, so I shifted my focus more towards making something to work fast.

2.2 The Original Idea

In the beginning, I divided the problem in two phases. First, I wanted to model the WiFi sensors, i.e. have a model per sensor where each WiFi source would have probability distributions. Then, my goal was to infer the likelihoods of each models based on the observations and continue to infer the likelihoods of each value in the discrete [0-99] distribution. I prefer to get something working as fast as possible, so I went with k-means clustering, which is probably the simplest possible way to divide the training data to n clusters. After this, I made the assumption that a sensor's strength value distribution for a single WiFi source should be the following

1. Single separate value for zero strength. WiFi source might be e.g. offline sometimes and provide a peak zero.

2. Normal distribution for other values. It might not be perfect, as there might be couple of different peaks for example, but to me it was the simplest reasonable model and thus a perfect way to start.

Clustering was working OK, at least speed-wise yet anything with Figaro seemed to be too slow. In fact it looked like the data size couldn't afford more than the simplest model. I decided to give it just that. Given that the WiFi sources would be independent from each others, Naive Bayes style model would actually be correct, and inference cheap. Even if the assumptions weren't going to be completely true, such model would be a way to produce results.

On the last minute I ended up trying something even simpler. I ditched the normal distribution and used discrete distribution instead, e.g. frequency with smoothing for each WiFi strength value [0-99]. This is motivated by the idea that normal distribution is too simple model for the amount of data we have and loses some predictive power.

2.3 Limitations of the Model

The model is far from perfect. To my mind, one issue is the assumption that the WiFi sources would be independent from each others. This is clearly wrong, as if we have two WiFi sources S_1 and S_2 in the same room, having a strong reading from S_1 indicates a reading from the other as well. This is likely to show up in the results as overconfidence for the sensor choice, as if we have a reading of S_1, we should give less weight for S_2 as an evidence.

Sensors are originally expected to be uniformly distributed, which probably is not the case. If one sensor has in reality more readings than the other, the model presented here won't take it into account.

2.4 Ideas for Improvement

My money for the biggest single improvement would go to releasing some of that "WiFi sources are independent from each others" assumption. Having a complete Bayes network would be computationally way too expensive, but the most obvious connections should be modeled.

Another minor thing is that the clustering results should be possible to be saved, so it would be easier to focus on the development of the latter part of the algorithm.

3 Results

The target minimum score was -22700, which is a log likelihood so a larger, or less negative, is better. First I used only fraction of the training data to get results out faster. I tried couple of things. First scores with 8000 training samples out of the 100000 and around 100 sensors provided scores around -21500, which was at least a bit better than the minimum. Increasing sensor count to 200 didn't make a big difference, best score with this limited training data was -21417.

When using all the training data, the result got worse. This is because of the nature of smoothing, combined with the problematic sensor independency assumption. This indicates that a better score could be achieved quite easily with trial and error, but what really would be needed is the proper conditional probabilities.

Not using the normal distribution seemed to help things a lot. With training size of 8000, the first result I got was -16912. After using whole data set for training, result improved to -14883. This was with having 100 clustered sensors in the background.

4 Running Instructions

The program is built with Scala 2.11.8 and sbt 13.11. sbt assembly plugin is used to make an executable jar, so building a new executable and running the scoring happens with the following terminal command from the project directory:

```
sbt assembly && python ScoringProProMo/scoring.py
```

The code can be found from Github, `https://github.com/juhokallio/WiFi-Predictor`.