# Capstone Project Final Report

Massimo Morelli
Friday, October 10, 2014

## Introduction

Natural Language Processing (NLP) is a field of computer science, artificial intelligence and linguistics concerned with the interactions between computers and human (natural) languages ([1]). In the contest of NLP a language model is defined as a probability distribution over strings that describe how often each string occurs as a sentence in some domains of interests ([2]). Much research has been carried out in this field ([2]) ,([3]) ,([4])

The goal of my model, i.e. the goal of the Capstone Project, is to determine the next word given the preceding words in a phrase. This could be for example useful in texting application for smart phones. The simplest way to do this is to compute a Maximum Likelihood (ML) estimate, in which the probability of a word sequence is the relative frequency of the word sequence in some corpus of training data.

Having the probability of all sequences of N words (N-grams) could permit to calculate exactly the probability of the next word, simply using the ML estimate. Unfortunately in real world application it is very rare to have all sequences of words available. It can happen that we were given a sequence that never occurred in training. In this case a ML model could not give a prediction.

To address this problem a class of techniques called Smoothing is employed. The purpose of Smoothing is adjusting the ML estimate of probabilities to produce more accurate probabilities. These techniques tend to make distributions more uniform adjusting low probability such as zero upward and higher probability downward. Many smoothing techniques have been proposed ([2]), among other the simple Good Turing (which tries to estimate the never seen N-gram using the probability of the once-seen), Katz smoothing (which add the combination of higher order and low order model), and the Kneser-Ney smoothing which I used and I will describe very briefly. I reference to ([2]) and especially ([3]) for an extensive description and example of implementation.

## The Kneser-Ney model

Kneser-Ney, as many other Smoothing techniques, uses the lower N-gram probability to complement the higher N-gram. However, a lower order distribution is a significant factor in the combined model, only when few or no counts are present in the higher order distribution (we look up in N-gram only when we have failed to find an N+1-gram). What is relevant, in the lower order distribution, is the continuation probability (i.e. how many different words could follow this lower level N-gram) more than the ML probability (i.e. how often this lower level N-gram occurs). So a KN model combines the ML probability of the maximum N-gram utilized (in my case trigram) with the continuation probability of lower level N-gram (bigram and unigram). A lump of probability is reserved for lower level N-grams discounting the maximum level probability with a fixed quantity D.

Using mainly ([3]) as a reference I developed a trigram KN model, using the following equation:

$$P_{KN}(w_3 \mid w_1 w_2) = \frac{max\{c(w_1 w_2 w_3) - D, 0\}}{c(w_1 w_2)} + D * \frac{\mathrm{N}(w_1 w_2 \bullet)}{c(w_1 w_2)} * \left( \frac{max\{\mathrm{N}(\bullet\, w_2 w_3) - D, 0\}}{\mathrm{N}(\bullet\, w_2 \bullet)} + D * \frac{\mathrm{N}(w_2 \bullet)}{\mathrm{N}(\bullet\, w_2 \bullet)} * \frac{\mathrm{N}(\bullet\, w_3)}{\mathrm{N}(\bullet\bullet)} \right)$$

This is the model I used for my predictions.

## The given data

The biggest problem in this project is to cope with the amount of data. Languages are extensive, so we need big corpuses (collections of text). Those that are available to us are not so big (in a "big data" sense) but are a significant challenge for a single PC (especially mine, with only 4G RAM). That required an extensive experimentation. The three files available have the following dimension (a line of the input file is a document):

| File | Documents | Words |
|---|---:|---:|
| en_US.blogs.txt | 899,288 | 37,919,962 |
| en_us.twitter.txt | 2,360,148 | 31,176,356 |
| en_us.news.txt | 77,259 | 2.751.816 |

Words frequency follows as expected Zipf law (I have explored this theme deeply in the status report in mid project). Some facts worth noting: in a sample of 100,000 documents we have about 125,000 unique words with 3,100,000 word instances. In this sample 70,000 words compare only once. In Natural Language Processing parlance, these are Hapax Legomenon or Singleton.

## Model, algorithm and code

All code for available on github [5]. The functions here employed are loaded from github with the source command.

```r
Github_URL <- "https://raw.githubusercontent.com/momobo/Capstone_Dryrun/master/code/CapstoneBase.R"
library(devtools)           # this is a hack to solve source problems ... see note 1.
source_url(Github_URL)      # ... with https on windows (thanks stack overflow)
```

The training of the model begins with the division of the corpus in training, validation and text. The proportion I chose is 60%, 20%, 20%.

```r
# divides between train, validation, test
partitionData(datadir, fileb)
```

Then I could start loading the train set to train the model. After some test I determined that (performance wise) the optimal piece of corpus I could load is 2,000 documents from the blog corpus. I proceed to break the corpus in phrases, extracting more phrases from each document, simply breaking on stop words (points, question and exclamation marks). I also mark beginning and end of phrases with special marks. In this phase I also performed some minimal cleaning and filtering activities on the corpus. I do not apply stemming, as I think it is not useful in this project. Also bad words are not filtered (it does not pose technical difficulties, if you search in the code the elimination of bad words is just out-commented) but I left it out from the project as this is not going to be a commercial app where someone could be offended and sue).

```r
# create partitioned corpus
corpfiles <- createCorpus(trainfile, limit=limit)
# save partition index
save(corpfiles, file=paste(datadir, CFILES, sep="\\"))
```

After this phase, I have 270 file in my working directory, each representing 2,000 sentences of the original corpus. The next phase is load and tokenize the corpus in unigram, bigram and trigram.

The tokenization with lm() library is very slow and scale badly. My first test shows that I could not produce a term document matrix (TDM) of more than 10,000 documents. But with 2,000 documents it is possible and relatively fast. An important help comes from the fact that in the end I need only the relative frequency of the N-gram and not the full TDM matrix: i could ignore in which document the N-gram appeared. The loading is performed with the function `loadNGram()` that create a TDM and then convert it to a data table. The function `pasteNGram()` calls repeatedly `loanNGram()` and put together all the data table resulting. The use of data table is instrumental to the success of the training routines. Data table are much faster than data frame and the use of keys helps very much in the solution.

The load is driven from the `pasteNGram()` routine, that uses a vector (corpfiles) with the name of all the 2,000 documents pieces (there are 270 of them). The resulting data tables (unigram, bigram and trigram) are then saved on the project directory. The data table concerning bi and trigram are then enriched of indexes to manipulate them during the next phases. Also the start of phrase tags are now removed, as they are not counted in the smoothing algorithm (see note 4 in [2]).

```
#  create data tables
tddf  <- pasteNGram(corpfiles, 1, tolower=TOLOWER)
tddf2 <- pasteNGram(corpfiles, 2, tolower=TOLOWER)
tddf3 <- pasteNGram(corpfiles, 3, tolower=TOLOWER)

#  add index and remove start of phrase
tddf2 <- addIndexN(tddf2 ,2)
tddf3 <- addIndexN(tddf3 ,3)
tddf  <- removeStartOfPhrase(tddf,  1)
tddf2 <- removeStartOfPhrase(tddf2, 2)
tddf3 <- removeStartOfPhrase(tddf3, 3)
```

The core of the KN algorithm is implemented in the following routines (`addKNUnigram()`, `addKNBigram()`, `addKNTrigram()`). The KN formula above for the trigram probabilities could be considered recursively as follows:

$$P_{KN}(w_3 \mid w_1w_2) = P_{ML-discounted}(w_3 \mid w_1w_2) + \lambda(w_1w_2) * P_{cont}(w_2w_3)$$

$$P_{cont}(w_2w_3) = P_{cont-discounted}(w_2w_3) + \lambda(w_2) * P_{cont}(w_3)$$

The fastest way to calculate the probability is to pre-calculate the part that are only dependent from a word in the unigram data table and the part that depends from two words in the bigram data table. Then we could calculate the KN probability in trigram (and the continuation probability in bigram and unigram) *in just a single pass*, using lookup in the other data structures.

```
# calculate probability from the unigrams up
ntddf <-  addKNUnigram(tddf, tddf2, tddf3)
ntddf2 <- addKNBigram(ntddf, tddf2, tddf3)
ntddf3 <- addKNTrigram(ntddf2, tddf3)
```

Once I have the trigram PKN loaded in the trigram data table the probability became just a matter of checking the last trigram and taking the biggest KN probability. Reversing to the lower order N-gram a simple lookup gives the continuation probability that could be applied. In case of absence of information (typos or mangled or truncated words) a default (low) probability is employed.

The final model has the following dimension:

```
## unigrams:  345297 number of words in corpus:  23772767

## bigrams:   220707

## trigrams:  178411
```

The data tables generated in this way contains all the information needed to give the probability of the next word. They are also surprisingly compact, even without stripping them of the fields that are only necessary for the model construction.

The model is simple and fast to use. Just apply the `nextWord()` function with the trained data structure (this makes it simple to switch languages):

```
pred <- nextWord("I am the", ntddf, ntddf2, ntddf3)

print(pred)

## [1] "only"  "first" "most"
```

## Further notes

### Unicode particularities

Some of the corpus we have to test were not English, so the use of Unicode was mandatory. I loaded the file as UTF-8 and wherever possible I applied the Unicode conscious library `stringi` ([6]) instead of simple regular expressions. For example the `tolower()` routine in stringi deals with special german character as ß that is correctly transformed in "ss" .

### Simplifications

Giving the dimension of the training data I had to recourse to some simplification. The biggest was the renounce of Hapax Legomenon (or Singleton) in bigrams and trigrams. This was mandatory, as the number of bigram and trigram that compare only once grows very fast with the dimension of the corpus, and I was already at the limit of what the available resources could process. Also I decided to do not treat symbols and special characters, as this is a refinement that would have requested a lot of trying and error. I leave this refinement for future developments in this data product.

### Model selection

There are many parameters that could be optimized. The way to test the model is testing the perplexity on the validation corpus. Perplexity is a measurement of how well a probability distribution or probability model predicts a sample ([7]) I wrote a routine to calculate the perplexity and I tested it on the validation corpus. Here is an example with a small sample of the validation corpus:

```
# measure perplexity in a slot of the validation set. NN: length of the slot
perp1 <- measurePerp(NN=100, slot=1)
print(perp1)

## [1] 1304
```

When applied systematically we could use this method to validate the model and to choose the value of parameters, as the discount value D. It is also important to note that it is possible to choose different parameter D1, D2, D3 for each level (unigrams, bigrams, trigrams). This would require much computing time and could be carried out in future research.

### Enhancements

I implemented some small enhancements to the model:

- When the last word is never seen (out of corpus word) there is no information in the KN model. In this case I implemented a backoff to a "jump one word": I look for the word before the never seen one and search between the trigram that begins with this word. For example if I want to predict after "new yokr" (a typo) I search between the trigram that begins with "new" as "new york times" or "new world order"," I jump the last (possibly mangled) word and give my prediction.
- German has very strict rules on capitalization of words. A substantive is always capitalized. I think that the overall prediction in German is better when I do not force the algorithm to lowercase as in English. A German person would always begin a noun with a capitalized word and this is important information for the model. So I made the capitalization a parameter of the model and I trained the german model without it.
- In my tests I also implemented a small world completion mechanism. I just propose the more likely unigram that begins with the characters typed.

## Possible future developments

There are a lot of things that could be tried:

- I am not satisfied with the amount of test I have been able to do. The model could be tuned to obtain better prediction. What is needed is a way to check faster different combination of parameters or just perform repeated simulation. A framework for fast machine learning would be extremely useful.
- Extended KN models ([2]) are analyzed in literature. They use different discount at each level. It is said that they perform better. But this would mean more parameter to tune.
- Having a powerful computer available for training could permit to remove the simplification of eliminating Hapax in multigrams (this could be tried just modifying a model parameter).
- It could be fruitful to correct mangled or misspelled word before to try the prediction. This strategy would need extensive tests as it could backfire easily.
- Word prediction and word completion are not used together in this model. This is an unnecessary limitation. I did not find any research that combine the two but I am reasonably certain that commercial applications use this important information to yield better predictions.

## Conclusions

The objective of the project was to build a prediction engine in two languages (English and German). The prediction engine has been implemented, tested and deployed (on Shinyapp.io([8])). Result are not comparable to a commercial application but they are positive. I choose to implement a KN model, more complex than a Good Turing one and I get it to work in a cheap PC. Once the model is trained, the prediction is fast.

## References

- [1] NLP http://en.wikipedia.org/wiki/Natural_language_processing
- [2] Chen Goodman http://www2.denizyuret.com/ref/goodman/chen-goodman-99.pdf
- [3] Martin Körner Thesis http://mkoerner.de/media/bachelor-thesis.pdf
- [4] Jurafsky complete course https://www.youtube.com/playlist?list=PL6397E4B26D00A269
- [5] Github code https://github.com/momobo
- [6] stringi package http://cran.r-project.org/web/packages/stringi/stringi.pdf
- [7] Jurafsky Perplexity video https://www.youtube.com/watch?v=OHyVNCvnsTo
- [8] Deployed application: https://momobo-coursera.shinyapps.io/CapstoneApp/