Word Prediction: Exploratory Data Analysis

Wei Xu

February 10, 2017

Data loading and cleaning

We load a sample corpus of txt documents composed of blogs, news and tweets provided by swiftykey in the data science capstone project. We used **tm** package to manage documents, which are loaded from a function called **Corpus**. In order to save memory, **PCorpus** (named permanent corpus) is used, which essentially creates pointers to external data base and manipulates underlying corpus. Several examples of individual entries in the corpus are shown below:

```r
library(tm)
library(filehash)
fname <- "../../final/en_US/"
# load English txt from blogs, news and twitters
docs <- PCorpus(DirSource(fname, encoding="UTF-8", mode="text"),
                readerControl=list(language="en"),
                dbControl = list(dbName = "docs_origin.db", dbType = "DB1"))
viewDocs(docs, 2, 5) # news example
```

```
## And when it's often difficult to predict a law's impact,
## legislators should think twice before carrying any bill. Is it
## absolutely necessary? Is it an issue serious enough to merit their
## attention? Will it definitely not make the situation worse?
```

Given the messy nature of language documents, especially txt data from blogs and twitters, it is important to first clean the original txt data in order to produce programmable dataset. First of all, **non-ASCII symbols**, which appear most frequently in twitter posts, should be removed from the original corpus.

Considering the spirit of word prediction, the words themselves are more important than their corresponding cases. Thus we conclude that our model is not case-sensitive and we generally transfer all words to their **low cases**. **Numbers** although appear so frequently in communications, are not within our interest simply due to their unpredictability. Whether or not to include **punctuations** as words in natural language processing is an under-debated question. Since punctuation is critical for finding boundaries and sentiments (comma, colon, period, question and exclamation marks), these punctuation symbols are treated as the same way as individual words. The apostrophe as a special punctuation, is often used to create progressive forms, contractions and plurals. In these cases, i.e. apostrophes are embedded into words, we treat the entities (apostrophe word combination) as individual words. Punctuations in other cases are considered unimportant and thus removed during the cleaning process.

Stopwords (words with little meaning, such as *and*, *the*, *a*, *an*), **stemmers** (inflected words with same stem/lemma but different wordforms) and **disfluencies** (word fragments and filled pauses, such as *uh*) are important concepts in natural language processing. We assume that all words with different wordforms, regardless of stopwords or stemmers, will be included in our model. Disfluencies, though may help in parsing (such as fillers), could be removed for a neater model. We also want to remove the **whitespace** in the end. **Duplicated pattern** appears quite often in twitter or blog posts. This duplication happens in both punctuations and words. Since this could be treated as a common language phenomenon, we choose to keep it in the cleared documents. Part of the unwanted words are transformed into symbols such as `<num>`, `<url>`, `<email>` and `<eos>`, and are removed after the generation of n-grams. We have a few examples of the documents after data cleaning:

```r
# load cleaned dataset
library(tm)
library(filehash)
cname <- "../clean/"
docs <- PCorpus(DirSource(cname, encoding="UTF-8", mode="text"),
                readerControl=list(language="en"),
                dbControl = list(dbName = "docs_clean.db", dbType = "DB1"))
# view transformed content
viewDocs(docs, 2, 5) # news example
```

```
## and when it's often difficult to predict a law's impact ,
## legislators should think twice before carrying any bill . is it
## absolutely necessary ? is it an issue serious enough to merit
## their attention ? will it definitely not make the situation worse
## ?
```

It is important to take an overview of the text data before applying it to build our text prediction model. A exploratory table of words and vocabularies are shown below. It is obvious to see that we have over 100 million words in our original complete dataset, which is really a huge amount of data. Total number of words in each genre of the documents (blog, news and tweet) are comparable to each other, while the corresponding number of lines are vastly different. Type/token ratios (TTR), also vocabulary/words ratio, are all around one percent, with TTR for tweets slightly higher. TTR and the diversity measure are two ways of measuring the complexity of each genre. Since the content of tweets is less restricted, thus it is consistent with the result of higher TTR and diversity.

```r
library(tau)
library(xtable)
ngram <- function(corpus, n) {
      textcnt(corpus, method = "string", n = n, split = "[ ]+", decreasing = TRUE)
}
# lines in corpus
line.blg <- length(as.character(docs[[1]][[1]]))
line.new <- length(as.character(docs[[2]][[1]]))
line.twt <- length(as.character(docs[[3]][[1]]))
total.line <- c(line.blg, line.new, line.twt)
# analysis of words in corpus
word.blg <- ngram(docs[[1]][[1]], 1)
word.blg <- data.frame(words = names(word.blg), counts = unclass(word.blg))
word.new <- ngram(docs[[2]][[1]], 1)
word.new <- data.frame(words = names(word.new), counts = unclass(word.new))
word.twt <- ngram(docs[[3]][[1]], 1)
word.twt <- data.frame(words = names(word.twt), counts = unclass(word.twt))
unique.word <- c(dim(word.blg)[1], dim(word.new)[1], dim(word.twt)[1])
total.word <- c(sum(word.blg$counts), sum(word.new$counts), sum(word.twt$counts))
summaryCorpus <- data.frame(Words = total.word, Lines = total.line,
                            Vocabulary = unique.word,
                            typeTokenRatio = unique.word / total.word,
                            Diversity = unique.word / sqrt(2 * total.word))
row.names(summaryCorpus) <- c("Blogs", "News", "Tweets")
print(xtable(summaryCorpus, digits = 4), comment=FALSE)
```

	Words	Lines	Vocabulary	typeTokenRatio	Diversity
Blogs	41914845	899288	385831	0.0092	42.1404
News	39127577	1010242	324216	0.0083	36.6503
Tweets	35660645	2360148	382462	0.0107	45.2875

Data splitting

Before developing models, we first split the original dataset into training (60%), testing (20%) and an additional development testset (**devset** 20%). Individual entries in the documents are randomly permuted before splitting. We word further on the training set to develop our word prediction model.

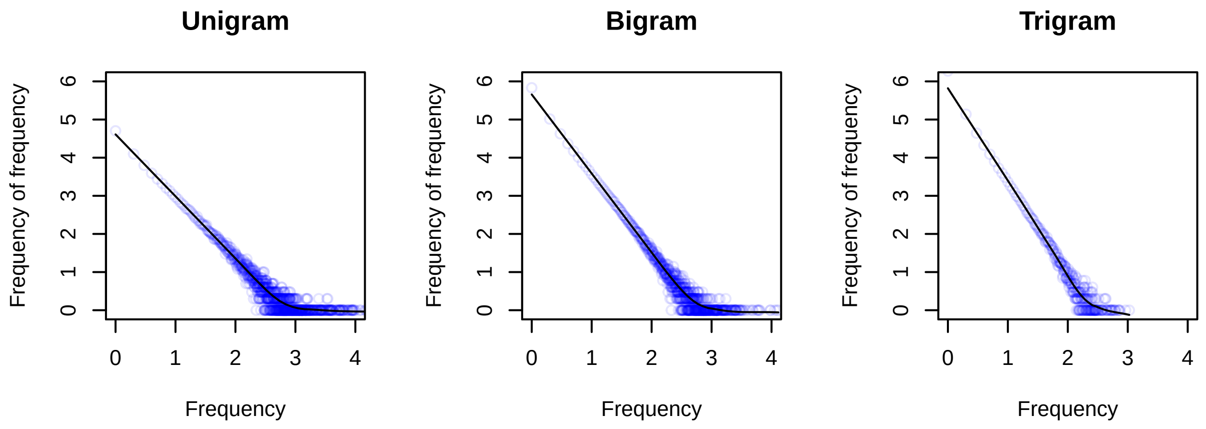
Generating n-grams

There are several ways to generate n-grams for our model prediction, including *ngram* package, *textcnt* function in *tau* package and *rweka*. The *tm* package supplies the most convenient method, given its built-in **TermDocumentMatrix** function. With customized tokenizer, one can essentially construct ngrams to any order. Unfortunately, the **TermDocumentMatrix** becomes extremely slow with massive dataset, prohibiting the practical usage of this method. Here, for the initial process of developing our model, we choose to use a pretty small dataset, with only 5% of the training data. Example code of creating unigrams is shown below. We also generate bigrams and trigrams in a similar way.

```r
unigram.tdm <- TermDocumentMatrix(docs.trial)
unigram.freq <- sort(rowSums(as.matrix(unigram.tdm)), decreasing = TRUE)
unigram.wf <- data.frame(words = names(unigram.freq), freq = unigram.freq,
                         row.names = NULL, stringsAsFactors = FALSE)
# clean abnormal words in unigram
# concatenated words, e.g. state-of-the-art
cnword <- grepl("^([a-z]+-)+[a-z]+$", unigram.wf$words)
# too long and rare words
lnword <- nchar(unigram.wf$words) > 12 & unigram.freq <= 2 & !cnword
unigram.wf <- unigram.wf[!lnword, ]
# words as <num>, <eos>, <url>, <email>
spword <- grepl("<(.*)+>", unigram.wf$words)
unigram.wf <- unigram.wf[!spword, ]
# long duplicate pattern, e.g. aaahhhhhh, ahahah, loooook
# grep of this pattern is hard to deal with long word string
dpword <- grepl("([a-z]+)+\\1{2,}", unigram.wf$words)
unigram.wf <- unigram.wf[!dpword, ]
# words start with apostrophe, e.g. 'stoke
apword <- grepl("^\\'", unigram.wf$words)
unigram.wf <- unigram.wf[!apword, ]
# deal with concatenate sign (i.e. -), all should be removed except concatenated words
cnword <- grepl("^([a-z]+-)+[a-z]+$", unigram.wf$words)
cnsign <- grepl("-", unigram.wf$words)
unigram.wf <- unigram.wf[!(cnsign & !cnword), ]
# write to folder
if(!dir.exists("../trial-ngram/")) dir.create("../trial-ngram/")
write.csv(unigram.wf, "../trial-ngram/unigram-wf.csv", row.names = FALSE)
# build frequency of frequency table
unigram.fft <- data.frame(table(unigram.wf$freq),
                          row.names = NULL, stringsAsFactors = FALSE)
colnames(unigram.fft) <- c("freq", "freq.freq")
```

```
write.csv(unigram.fft, "../trial-ngram/unigram-fft.csv", row.names = FALSE)
```

Although our initial trial dataset composed of only a tiny portion of the original dataset, it turns there is already approximately 100,000 distinct words in our trial set. The frequency of words decays exponentially, which means vast majority of the words simply appears only once (singleton). The proportion of singletons increases as we calculate higher order ngrams. For the first trial of the modeling, we choose to keep all the ngrams given the idea that all the sparse ngrams supply clues to the probability of unseen ngrams. This is the benchmark for the development of Good-Turing smoothing that is used later in our first model. The following figure indicates another interesting property of the frequency distribution, that is the frequency of frequency (N_c) decreases algebraically with frequency c. Here, N_c means the number of ngrams that occurs exactly c times. This algebraic relation serves as the ground to approximate the zero N_c in the data with a linear regression in the log space.



Good-Turing smoothing

The idea of smoothing comes from the reality that all the real text data is sparse. For ngrams that appeared sufficient number of times in the training set, one can obtain a fair estimation of the probability based on maximum likelihood estimate (MLE). However, for ngrams with zero or negligible probability, MLE can only give poor estimates. The intuition of Good-Turing smoothing is to use the count of singletons to help estimate the count of ngrams that never appeared. The Good-Turing discount d_c is defined as the ratio of the smoothed count with respect to its original count $d_c = c^*/c$. The left-over probability (or missing mass) for things with zero count now is given by $P^*_{GT}(N_0) = N_1/N$. The table below shows the results of discounts d_c for all ngrams that we built in our first model. One can observe that ngrams with lowest frequency got more discount, while the discount becomes smaller as frequency increases.

```
dirtrialngram <- "../trial-ngram/"
unigram.fft <- read.csv(file = paste(dirtrialngram, "unigram-fft.csv", sep = ""))
bigram.fft  <- read.csv(file = paste(dirtrialngram, "bigram-fft.csv" , sep = ""))
trigram.fft <- read.csv(file = paste(dirtrialngram, "trigram-fft.csv", sep = ""))
# implement Simple Good-Turing (smoothing zero Nc)
# a linear regression fit log(Nc) ~ log(C) is omitted given enough data
# discounted estimates used for counts c <= (k=5)
# custom function extended ngram.fft table with discount added
createNgramFftExtended <- function(ngram.fft, k=5) {
      ngram.fft$discount <- rep(1, nrow(ngram.fft))
      # calculate discount
      k = 5
      finalN <- ngram.fft[k+1,2]
      firstN <- ngram.fft[1,2]
      for(i in 1:k) {
            currN <- ngram.fft[i,2]
            nextN <- ngram.fft[i+1,2]
            currd <- (i+1.0)/i*(nextN/currN) - (k+1)*(finalN/firstN)
            currd <- currd/(1-(k+1)*(finalN/firstN))
            ngram.fft$discount[i] <- currd
      }
      ngram.fft
}
# apply to uni-, bi- and tri-grams
unigram.fft <- createNgramFftExtended(unigram.fft)
bigram.fft  <- createNgramFftExtended(bigram.fft)
trigram.fft <- createNgramFftExtended(trigram.fft)
write.csv(unigram.fft, "../trial-ngram/unigram-fft.csv", row.names = FALSE)
write.csv(bigram.fft,  "../trial-ngram/bigram-fft.csv",  row.names = FALSE)
write.csv(trigram.fft, "../trial-ngram/trigram-fft.csv", row.names = FALSE)
# print xtable of discount
library(xtable)
tableGT <- data.frame(freq=c(1:6), uni.discount=unigram.fft[1:6,3],
                        bi.discount=bigram.fft[1:6,3], tri.discount=trigram.fft[1:6,3])
print(xtable(tableGT), comment=FALSE)
```

	freq	uni.discount	bi.discount	tri.discount
1	1	0.33	0.23	0.12
2	2	0.68	0.58	0.47
3	3	0.76	0.71	0.64
4	4	0.84	0.77	0.71
5	5	0.89	0.81	0.77
6	6	1.00	1.00	1.00

Prediction with Katz back-off

The Good-Turing discount is usually combined with the Katz back-off method to give better prediction of words. For Good-Turing smoothing, left-over probability mass is assigned to all unseen events. Among those unseen events, the probability mass are assumed to be evenly distributed. As one step further, Katz back-off suggests a better way of distributing the probability mass among unseen events, by relying on information of lower-order ngrams.

Since our back-off algorithm involves calculating probability of suggested output word, we need to figure out what to put into the pool of suggested words and then select from them with highest probabilities. An ideal case is to try all the individual word in the whole vocabulary in our training set, which soon turns out to be a formidable task. Here, instead, we first use the naive back-off algorithm to select a pool of 50 words from the training set and then proceed with Katz back-off to find the final six suggestions of the next word. In detail, we select 20, 20 and 10 words with top effective probabilities from trigrams, bigrams and unigrams respectively. A few randomly picked examples of the prediction is given.

```
library(xtable)
print(xtable(nextWordPrediction("consider the following")), comment=FALSE)
```

	1	2	3	4	5	6
predictor	:	me	year	!	a	you
probability	0.12676056	0.10418327	0.08450704	0.05708672	0.05137805	0.05137805

```
print(xtable(nextWordPrediction("this is made in")), comment=FALSE)
```

	1	2	3	4	5	6
predictor	the	china	my	a	his	this
probability	0.21875000	0.04460415	0.04460415	0.02980817	0.02980817	0.01929384

```
print(xtable(nextWordPrediction("I'm running out of")), comment=FALSE)
```

	1	2	3	4	5	6
predictor	the	my	a	it	his	their
probability	0.27522350	0.04916986	0.04278416	0.02809706	0.02618135	0.02618135

Further discussion and consideration

It is kind of amazing that our model already gives some reasonable prediction of the next output word, given the small trial training dataset we used. There are still many future challenges. One of them is to **increase the training dataset** to 60% (or higher) of the original text set to increase prediction accuracy. The subsequent issue with **scalability** should be considered seriously. The **Katz back-off algorithm** implemented here, though gives reasonable prediction of next words, is still pretty slow given the size of ngrams created. Improvement on the algorithm is desired.

One additional issue that might deserve to mention is related to a simple and involved example. Consider the occasion that, the first $(n-1)$ words from the input data is not found in any of the ngrams (for $n > 1$). Then we have to go back to the case with unigrams and output a word with highest discounted effective probability, which is independent of any input words. This is definitely something unwanted. To get rid of the issue, an algorithm, which could further understand the intrinsic meaning (or **semantics**) of words or sentences is highly desired. Of course, a word that absent from the vocabulary of the training set can never be predicted, which is another limitation of our model.

References

Jurafsky, D. & Martin, J.H. (2000). *Speech and language processing: An introduction to natural language processing, computational linguistics and speech recognition*. Englewood Cliffs, NJ: Prentice Hall.

Gendron G R. *Natural Language Processing: A Model to Predict a Sequence of Words*. MODSIM World 2015, 2015, 2015 (13): 1-10.

Thach-Ngoc TRAN *Katz's Backoff Model Implementation in R*.