Milestone Report

Coursera student 75055

28 december 2015

.

1- Introduction

This paper gives a concise impression of my approach to the December 2015 capstone project for the data science specialisation.

.

2- Understanding the problem

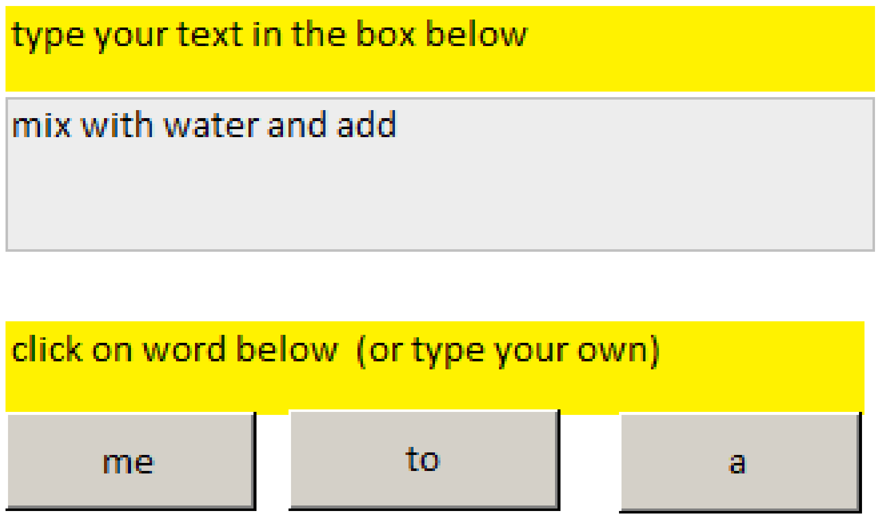
problem context

Typing text on modern mobile devices like smart-phones and tablets is an awkward and slow process. To speed things up swiftKey has developed an app. This app scans the text the user entered thus far and tries to predict which word the user will type next. The app gives the user three suggestions for the next word that are most likely. The user can choose the correct word from these three (or type an other word).

SwiftKey is interested in suggestion for improving the prediction algorithm. In the capstone project we are challenged to experiment with this kind of algorithms.

goal

The result of my project should be a shiny app that looks like this:


type your text in the box below
mix with water and add
click on word below (or type your own)
me
to
a

The user can type a text in the grey box. Each time the user types a space (I.e. ends a word) the app gives three suggestions for the most likely next word. The user can add the word to the text by clicking on the appropriate button.

problem

Given the above; how can we predict the three most likely possibilities for the next word?

constraints

Any acceptable solution to the stated problem should respect the following constraints:

- the results should be produced within 100 milliseconds (I.e the algorithm should execute fast on a smart-phone or tablet.) So computing intensive algorithms that require advanced hardware are not an acceptable solution.
- the apps should not require more than 50 MB of working memory (I.e. lookup tables should have a limited size),

measuring success.

Conforming to good data science practices we will use a test set of sentences. The percentage of test set words that are predicted correctly will be the used as a measure of quality.

Note that from the perspective of the user this measure is potentially problematic. The user is interested in speeding up the typing process. If the algorithm mostly suggest short words, little time will be gained. For the user the success is measured by the time saved.

.

3- Technical note

(important info for programmers, the general public may skip this section.)

I will use Perl to prepare the data, do the analysis and construct the lookup tables.

- Perl can read files line by line. Even very large files can be processed (read one line from disk, process this one line in RAM, write the result to disk, read the next line, etc.). The size of the files is limited by the hard-disk size, not the RAM size.
- Perl has good facilities for handling text (e.g. an excellent regular expression implementation).
- Perl has native and efficient hashes (I.e. excellent for implementing lookup tables).

The Perl results will be imported into R by using .cvs files.

The main drawback is that I can not show my Perl programs in a R environment. So, here I will only present the results (see my github repository for the programs and the csv files). An other drawback is that it is difficult to keep track of my work-flow (producing basic bi-grams takes about 20 steps).

.

4- Data acquisition and cleaning

Data acquisition and selection

The capstone project provides us with three files that can be used.
The basic summaries of the data, such as word and line counts, of these three files are:

- en_US.blogs.txt (210 MB, 899,288 lines of text, about 38,000,00 words)
  harvested from blogs on the internet.
- en_US.news.txt (206 MB, 1,010,242 lines of text, about 35,000,00 words)
  harvested from news feeds on the internet
- en_US.twitter.txt (167 MB, 2,360,148 lines of text, about 30,000,000 words)
  harvested from twitter messages on the internet

That is a large amount of data, there is no need to collect more. However we should not only have enough data, the data should also be representative of the type of text we want to write. Different users use words differently, furthermore the usage of words depends on the type of text an user is composing (a love tweet uses words differently than an academic article). So in an ideal world we would use a corpus that is specific for the user and for the type of text.

It is clear that the twitter corpus is most representative of the type of texts swiftKey users tend to type (relative short messages with in-official but "efficient" English with lots of LOL, u, r and 4play). So I decided to use the Twitter corpus solely.

A Perl program assigned Half (50%) of the lines in the Twitter corpus randomly to the train set (twit_train01.txt). The other half was assigned to the test set (twit_test01.txt).

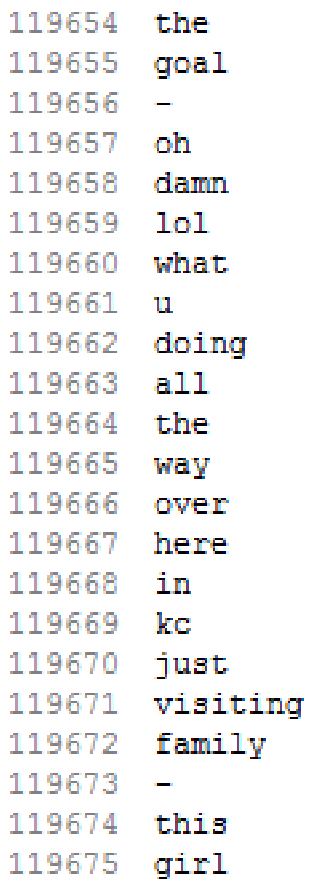
Data cleaning and transformation

Different Twitter users have very diverse ideas about the proper way capital letters should be used. To normalize this, all letters were converted to lower case.

All characters except the lower-case letters a,b,c…z, the space and the single quote ' were removed from the text files.

A line is split up into words by splitting on spaces. Note that under this rule i'm is one word.
To make processing in Perl easier, each word was put on a separate line.
The end of a sentence was represented by a line with only a - on it.

The figure below shows an illustrative fragment of the cleaned and transformed twit_train01.txt file.

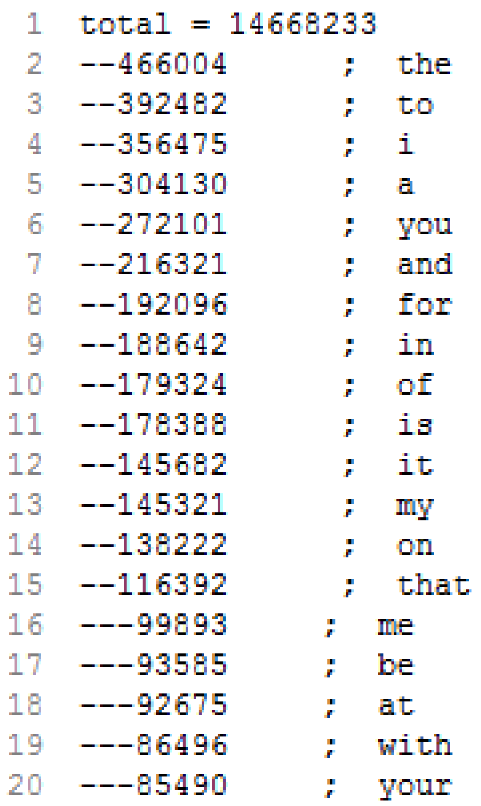


```
119654  the
119655  goal
119656  -
119657  oh
119658  damn
119659  lol
119660  what
119661  u
119662  doing
119663  all
119664  the
119665  way
119666  over
119667  here
119668  in
119669  kc
119670  just
119671  visiting
119672  family
119673  -
119674  this
119675  girl
```

.

5- Exploration

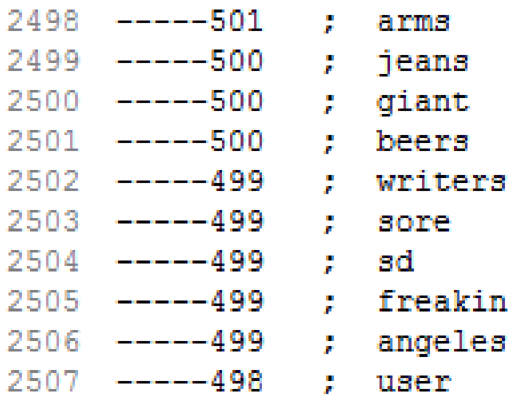
The words in the input file were counted, the counts were written to an output file (see below for a fragment).



```
 1  total = 14668233
 2  --466004      ;  the
 3  --392482      ;  to
 4  --356475      ;  i
 5  --304130      ;  a
 6  --272101      ;  you
 7  --216321      ;  and
 8  --192096      ;  for
 9  --188642      ;  in
10  --179324      ;  of
11  --178388      ;  is
12  --145682      ;  it
13  --145321      ;  my
14  --138222      ;  on
15  --116392      ;  that
16  ---99893     ;  me
17  ---93585     ;  be
18  ---92675     ;  at
19  ---86496     ;  with
20  ---85490     ;  your
```

So in total there are 14,668,233 words in the file; of which 466,004 are "the" and 392482 are "to", etc. In total, the corpus has 307,077 distinct words.

The frequency of occurrence of words drops exponentially. At line 2501 the frequency drops below 500.

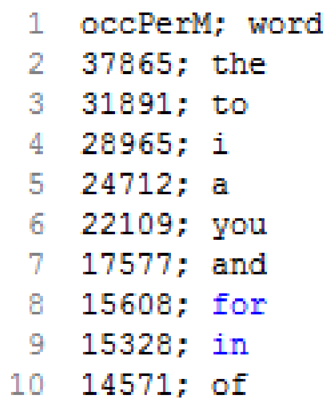


```
2498  -----501  ;  arms
2499  -----500  ;  jeans
2500  -----500  ;  giant
2501  -----500  ;  beers
2502  -----499  ;  writers
2503  -----499  ;  sore
2504  -----499  ;  sd
2505  -----499  ;  freakin
2506  -----499  ;  angeles
2507  -----498  ;  user
```

Below a frequency of 500 occurrences, a word is used very seldom. So an user almost never types these words, thus -on average- **little typing time is lost when we never suggest these words**. To keep the lookup tables small I decided to ignore these words in predictions. Note that this simple rule reduces the size of the lookup tables with 99%, with almost no costs in terms of user satisfaction.

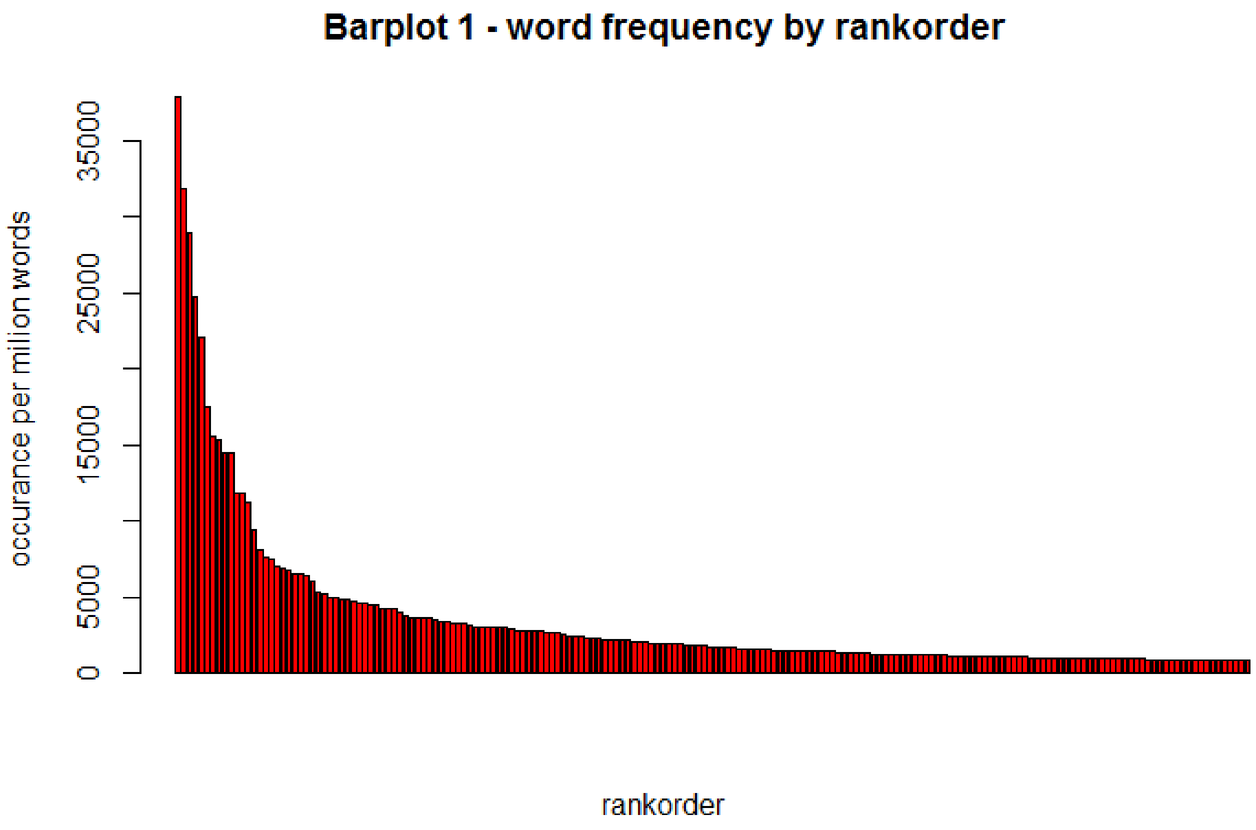
Technical note: when we represent a bi-gram as predictor->word; The ->words were selected from the +500 list, the predictors may be any word (I.e. are not limited to the +500 list).

The +500 words were written to file twit_selected_words.csv. In that file for each word the number of occurrences per million words (occPerM) was calculated (see below for a fragment).



```
occPerM; word
37865; the
31891; to
28965; i
24712; a
22109; you
17577; and
15608; for
15328; in
14571; of
```

Even in the 2500 most frequent words, the rate of occurrence drops exponentially. The bar-plot below shows the drop among the first 200 words (the figure for all 2500 words would have been almost L shaped).



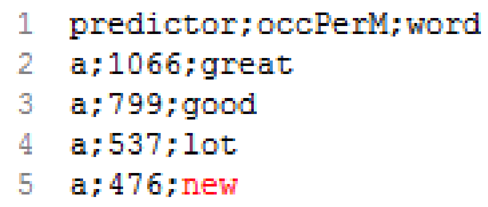
The bar-plot suggests that we could have limited the number of words even further (say to 500) with limited disadvantages.

It took 30 minutes to hand-scrutinize the list of 2500 words for bad language and proper names (wtf, fuck, fucking, damn, nigga, jeff, ron etc). 138 words were eliminated in this way. The remaining list of 2362 good words was written to a file goodwords.csv. Only words on the good words list will be suggested to the user (I.e. no bi grams will be constructed that produce these "bad" words).

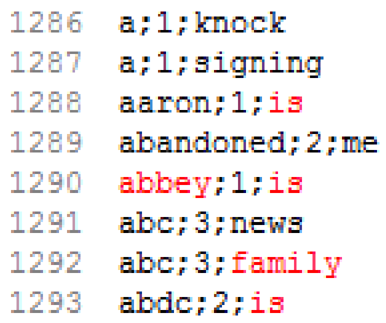
.

6- Modeling

Using the above mentioned files the next step is to list all the bi-grams predicting allowed next words. Two fragments of this list are shown below.

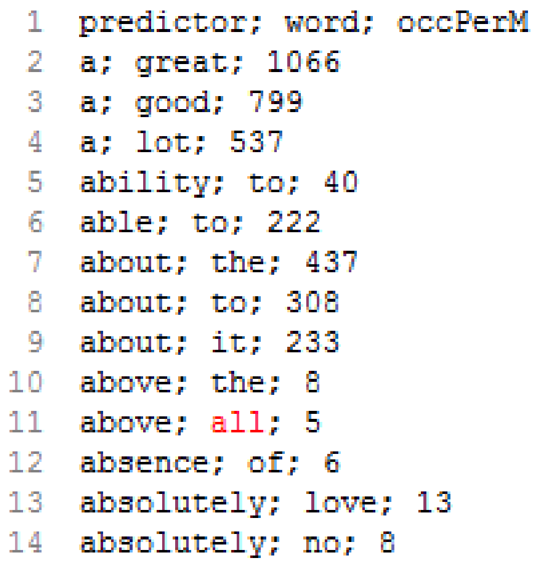


```
1  predictor;occPerM;word
2  a;1066;great
3  a;799;good
4  a;537;lot
5  a;476;new
```



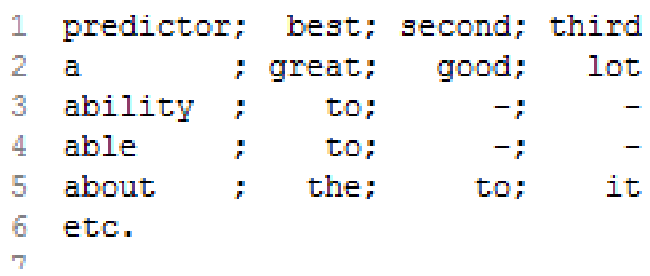
```
1286  a;1;knock
1287  a;1;signing
1288  aaron;1;is
1289  abandoned;2;me
1290  abbey;1;is
1291  abc;3;news
1292  abc;3;family
1293  abdc;2;is
```

The occPerM is the number of bi-gram occurrences per million bi-grams in this corpus.
Note that there are no less than 1286 bi-grams that start with "a", only the first three are useful for our purposes. An Perl program was used to select the three best bi-grams for each predictor. The result was written to a file twit_best_bigram.csv. The top of this list look like this:



```
1  predictor; word; occPerM
2  a; great; 1066
3  a; good; 799
4  a; lot; 537
5  ability; to; 40
6  able; to; 222
7  about; the; 437
8  about; to; 308
9  about; it; 233
10 above; the; 8
11 above; all; 5
12 absence; of; 6
13 absolutely; love; 13
14 absolutely; no; 8
```

The next step is to make one line per predictor by constructing a file like this.



```
predictor;  best; second; third
a        ; great;   good;   lot
ability  ;    to;     -;     -
able     ;    to;     -;     -
about    ;   the;    to;    it
etc.
```

For each predictor we can now lookup the three most likely next words.
This table can be used as a simple model of typical language use in Twitter-like texts.

.

7- Further development

The first thing on the to-do list is to create a shiny app that uses the simple model to predict the three most likely next words.
So my first priority is getting a shiny app working.

Next I could do some boring things like complicating the model by adding tri-grams and quad-grams and a back-off mechanism to choose between rivalling x-grams.

Alternatively I could do something useful or creative.

usefull Working with tools like Perl, to do the data wrangling, works very fine. But one important drawback is that my work-flow is badly documented (hence not reproducible). There are tools to fully automate - and document - these type of work-flows. It would be really useful to master this tools.
So I should repeat what I already did, but now using the tools promoted by the book: Data science at the command line.

creative Instead of tri-grams we could use shifted bi-grams (e.g. a bi-gram that does not predict the next word but the word after that).

Furthermore it would be interesting to think of a way to select the next word that is optimal from a user perspective. The user wants to save as much typing time as possible. Hence it is better to suggest a long word than a one letter word. That might be true even when the one letter word is slightly more likely. Is it possible to create a criterium that selects the next word that, on average, saves the most typing time?

.

Your advise and suggestions are welcome.
Is it acceptable to be non-boring?
If so, do you have other useful and creative suggestions?