Word Predictor

A Word Prediction Algorithm based on Stupid Backoff N-gram Model built within the partnership of Johns Hopkins with SwiftKey, the leading company on predictive text input for Android and iOS keyboards, to fulfil the Johns Hopkins Data Science Specialization Capstone Project requirements.

This README describes the algorithm and all files used to build and run this app.

Introduction

As people around the world are spending more and more time on their mobile devices for email, social networking, banking, and many other activities, any device that makes it easier for people to type is welcome. But typing on mobile devices can be an acute pain, and old systems, such as T9 and WordWise, sometimes produced hilarious "damn you autocorrect" results.

SwiftKey (http://swiftkey.com), the partner in this capstone, among other things, is a world leader in predictive text analytics. Their custom keyboard app on Android and the SwiftKey Note app on iOS are well-known. SwiftKey has been even integrating their predictive text technology into Professor Stephen Hawking's (http://swiftkey.com/en/giving-back/#hawking) existing system to improve his ability to communicate.

The goal of this Capstone Project involves developing a predictive text application that is capable of inferring meaning and intent from text the user types to predict his next word, very similar to Swiftkey's technology implemented in our present smartphone keyboards.

This more advanced, probabilistic-language-modeling-based approach 'knows' how certain words tend to be combined together in our language and tends to exhibit a far greater accuracy.

The application comprises of two parts:

- a prediction model algorithm, based on a predictive model of text starting with a vast, unstructured database of the English language, and
- a Shiny app UI that implements it.

In this capstone, Data Science will be applied to the area of Natural Language Processing (NLP).

In this report, we will highlight the results of some exploratory data analysis and detail the next steps to complete the application.

This work was done with the resources of the R statistical data analysis language (R Core Team, 2016) `r R.version` on `r OS.version` . All efforts were made to conform to the best practices of Reproducible Research (Peng, 2011, 2016a).

The following R packages were used for the data cleaning and analysis:

- *tm*, for text mining,
- *knitr*, for report generation (including nice tables),
- *stringr*, for text manipulation, and
- *RWeka*, for machine learning algorithms for data mining

Highlights

- **Speed**: The probabilities were all previously computed and are loaded before execution. The app looks down the word tables to instantly recover the most likely next word.

- **Versatility**: The algorithm handles many contractions used in Internet language: e.g. "I'll b there 2day" will be translated as "i will be there today."

- **Safety**: Profanities and bleeped words (e.g. 'f***' and 'f#@%' (mailto:'f#@%')) are removed from user input as were also previously removed from the tables.

- **Naturalness**: Stopwords, on the other hand, were left in, as they are present in ordinary language and could be the expected next input from a user.

Project Requirements

The goal of this project is to create a product to highlight the prediction algorithm built and to provide an interface that can be accessed by others.

For this project it was developed:

1. A R program to create the $1-$, $2-$, $3-$, and $4-gram$ probability tables .RData files, using the "Stupid Backoff" smoothing.
2. A Shiny app that takes as input a phrase (multiple words) in a text box input and outputs a prediction of the next word.
3. An R-Studio Presenter slide deck consisting of no more than 5 slides pitching the algorithm and app as if it were being presented to a boss or an investor.

This repository (see below) contains all the files necessary to satisfy the project requirements.

SwiftKey Data

The data comes from a corpus called HC Corpora (www.corpora.heliohost.org). See the readme file (http://www.corpora.heliohost.org/aboutcorpus.html) for details on the corpora available. The files have been language-filtered but may still contain a few foreign words. The data actually used for this project is available here (https://d396qusza40orc.cloudfront.net/dsscapstone/dataset/Coursera-SwiftKey.zip)

After unzipping, one finds that it consists of texts in 4 different languages: 1) German, 2) English, 3) Finnish, and 4) Russian. One also finds that these texts come from 3 different sources: 1) blogs, 2) news, and 3) Twitter feeds. For this project, however, we will be using only the English texts.

Pre-processing

Due to the size of the files, pre-processing, cleaning, and tokenising the entire data set would require either be unacceptable large amounts of RAM, time or machines. As a result, practitioners have chosen to use less data (Callison-Burch et al., 2012) or simpler smoothing methods, such as Brants et al. "Stupid Backoff" one (2007).

Thus, to reduce it, a sample is created by combining 1% samples from each text source. For reproducibility, a seed is set for the randomization algorithms involved in the sampling process.

Before removing punctuation, these texts must be broken into sentences, as it does not make much sense building prediction models across distinct sentences.

Now, the texts are converted to lowercase and cleaned to remove punctuation, slashes, numbers, extra white spaces, hashtags (#something), shouts (@somebody), e-mail addresses and URL's. Here, regular expressions (http://en.wikipedia.org/wiki/Regular_expression), combinations of literals and a rich set of metacharacters (http://en.wikipedia.org/wiki/Metacharacter), which are characters that have special meanings to a computer program, are useful to search through strings to identify specific patterns of interest that might be very hard to identify other way.

Common contractions were also converted, e.g. to transform “I’ll b there 2day” to “i will be there today.” The lists of contractions were obtained from the Snowball project (http://snowball.tartarus.org/algorithms/english/stop.txt) and from the ranks.nl site (http://www.ranks.nl/stopwords)

Profanities were also removed. The list of profanity words was obtained from the Shutterstock repo (http://github.com/shutterstock/List-of-Dirty-Naughty-Obscene-and-Otherwise-Bad-Words). Besides, bleeped words (e.g. ‘f—’, ’f***’, 'f#@%' (mailto:'f#@%')) were also removed by means of regular expressions searches.

Stopwords, on the other hand, were left in, as they are present in normal language and could be the expected next input from a user

After all this cleaning, we can proceed to prediction.

Text Prediction

In the field of Natural Language Processing (NLP), n-grams are sets of n co-occuring words within a given window in a sequence of text or speech.

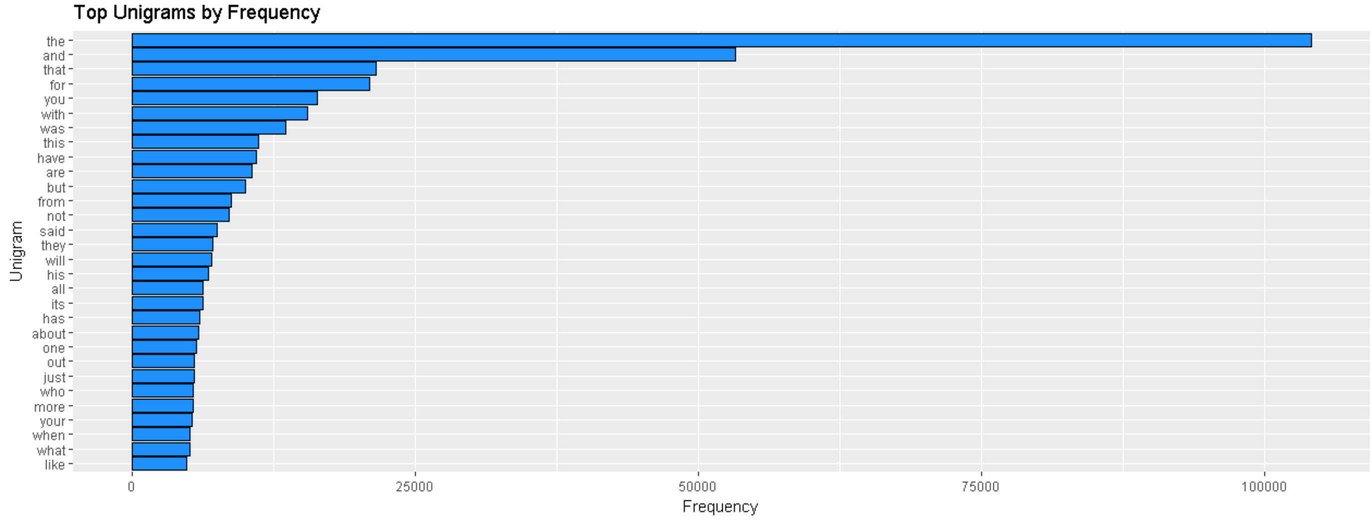
Stemming describes the process of transforming a word into its root form, such as ‘swim’ from the word ‘swimming’, which can, however, create non-real words. In contrast, lemmatization aims to obtain the canonical (grammatically correct) forms of the words, the so-called lemmas, and is, consequently, computationally more difficult and expensive than stemming. In practice, it was shown that both stemming and lemmatization have little impact on the performance of text classification (Toman, Tesar & Jezek, 2006) and, therefore, none of these processes was adopted here.

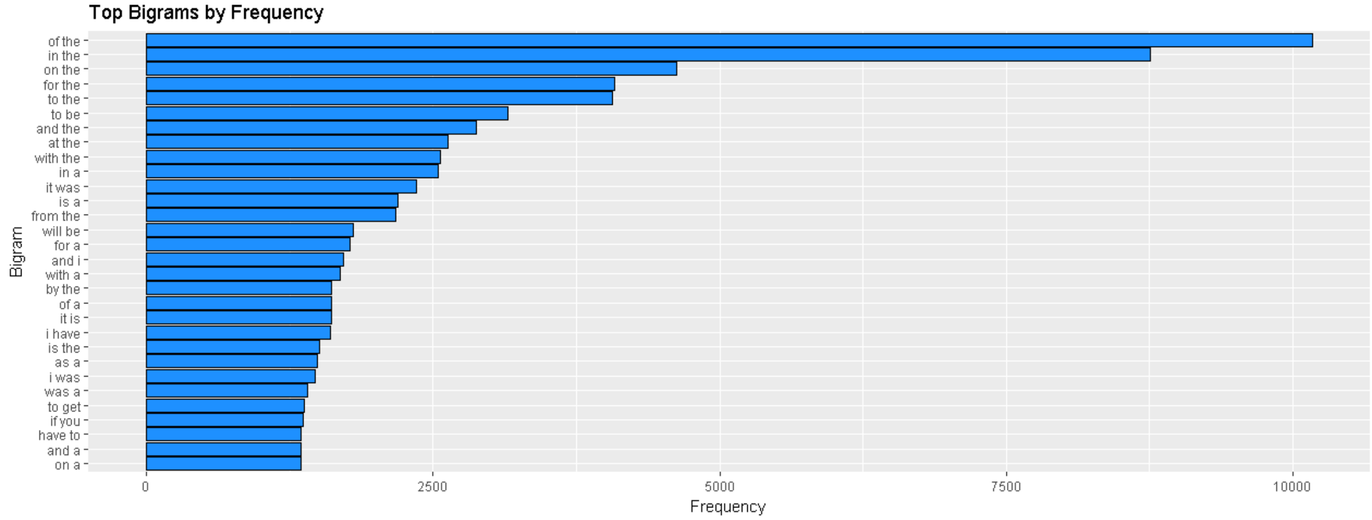
Here, we will calculate the frequencies of isolated words (1-grams) and 2-grams, 3-grams, and 4-grams, i.e. combinations of two, three, and four words in the dataset.

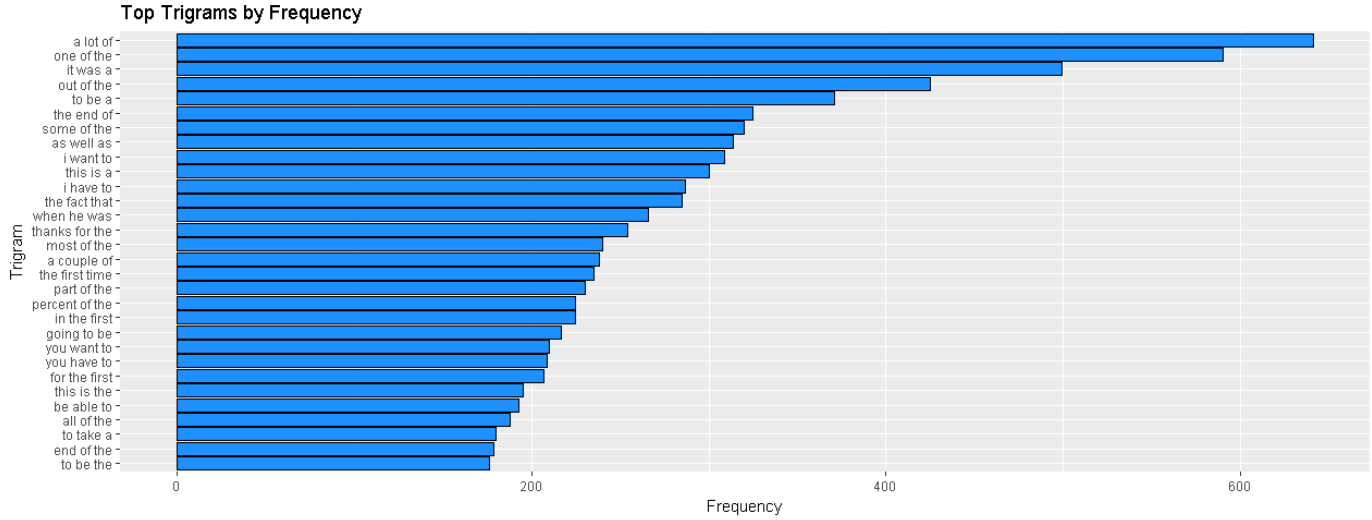
For that, we first constructed for the various n-gram tokens. This process resulted in dataframes containing the top n-grams with their corresponding frequencies in the text.

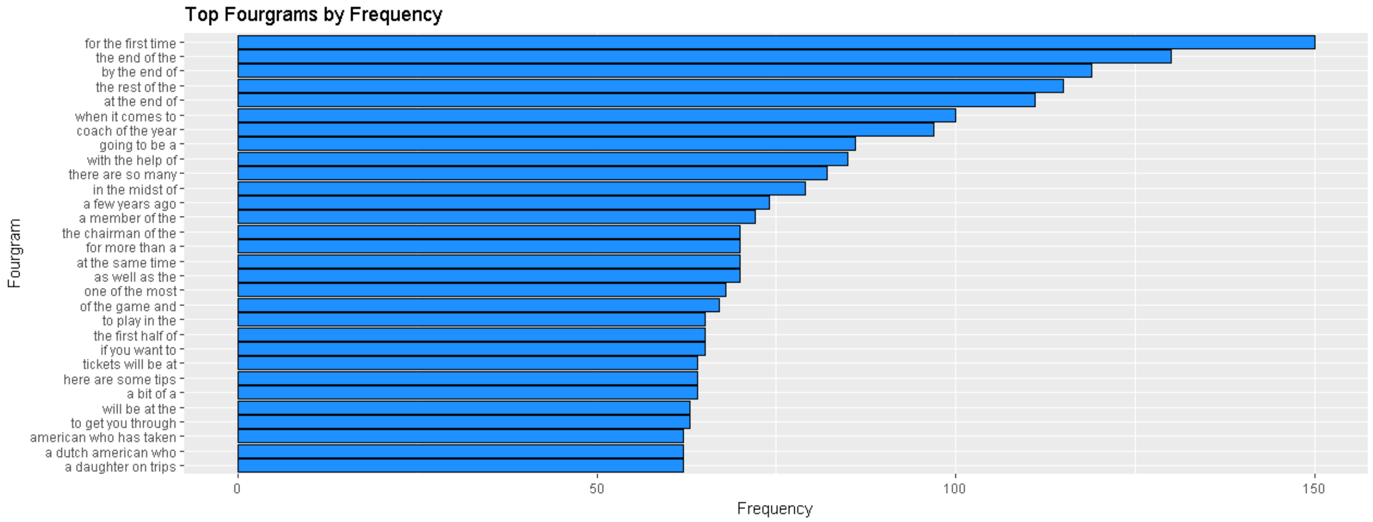
The proccess of n-gram creation resulted in many sparse terms, i.e., terms that rarely appear in the documents, which were then removed.

From the graphs below, one can see that ‘**the**’ and ‘**and**’ are by far the two most often occuring words, what is in accordance to their generalized use in normal speech. This also shows the appropriateness of our previous decision of not removing stopwords from the corpus, as doing so would have impacted our prediction model in a negative way.


Top Unigrams by Frequency
Unigram
the
and
that
for
you
with
was
this
have
are
but
from
not
said
they
will
his
all
its
has
about
one
out
just
who
more
your
when
what
like
0
25000
50000
75000
100000
Frequency


Top Bigrams by Frequency
Bigram
of the
in the
on the
for the
to the
to be
and the
at the
with the
in a
it was
is a
from the
will be
for a
and i
with a
by the
of a
it is
i have
is the
as a
i was
was a
to get
if you
have to
and a
on a
0
2500
5000
7500
10000
Frequency


Top Trigrams by Frequency
Trigram
a lot of
one of the
it was a
out of the
to be a
the end of
some of the
as well as
i want to
this is a
i have to
the fact that
when he was
thanks for the
most of the
a couple of
the first time
part of the
percent of the
in the first
going to be
you want to
you have to
for the first
this is the
be able to
all of the
to take a
end of the
to be the
0
200
400
600
Frequency



Having the n-gram x frequency look-up tables, we proceeded to develop a predictive model using them, combined with a back-off technique, to calculate the probability of the next word occuring.

- Step 1: The user input is filtered to remove profanity, contractions, punctuation, numbers, foreign characters, and any extra white space.
  - If the user enters e.g. “to be or not to” the algorithm chops it to the last 3 words (“or not to”).
- Step 2: The algorithm searches for a match to chopped input (“or not to _____”).
- Step 3: If a match is found, the algorithm skips to Step 4.
  - Otherwise, the user input is shortened again (“not to _____”, etc.), and the algorithm goes back to Step 2.
- Step 4: If a match is found, it is returned to the user interface.
  - Otherwise, the most frequent word in the corpora is returned.

One key aspect is the utilisation of “Stupid Backoff” approach (Brants et al., 2007). Specifically, at Step 3 above, if a match is not found the user input is shortened and this shorter version is searched again to increase the likelihood a match is found.

If the user provides random text to this prediction algorithm, e.g. “awpu1iub325 i1398th351bvnnd qwliwiu2451”, it just returns the most common unigram in the corpora. In this manner, it is avoided the use of extra memory while quickly returning a result.

Calculating Probability Scores

The idea here is the algorithm to find the word w, out of all possible candidate words, that maximises the probability that it is the intended next word, given the original sentence s, or

$$argmax_{w \in candidates}P(w|s)$$

.

Therefore, the algorithm would comprise four parts, corresponding to the terms of this expression:

1. A **Candidate Model** $w \in candidates$ that tells which candidate words w to consider.
2. A **Language Model** $P(w|s)$ that evaluates the probability that w would be typed in as the next word, given the sentence s. For example, $P(be|to)$ is relatively high, but $P(beeexyz|to)$ would be very low.
3. A **Selection Mechanism** $argmax$ that will choose that candidate w with the highest probability $P(w|s)$.

A (statistical) language model is one which assigns a probability to a sentence, which is an arbitrary sequence of words. Language modelling is used in speech recognition, machine translation, part-of-speech tagging, parsing, handwriting recognition, information retrieval and other applications.

By far the most widely used language model is the n-gram language model, which breaks up a sentence into smaller sequences of words (n-grams) and computes the probability based on individual n-gram probabilities. The intuition of the n-gram model is that instead of calculating the probability of a word given its entire history, we can approximate the history, in agreement with Markov's assumption (Markov, 1906), that only the last few n words are relevant when predicting the next one.

Now, given a large corpus of plain text, we would like to train an n-gram language model and estimate the probability for an arbitrary sentence, and any n-grams in a querying sentence which did not appear in the training corpus would be assigned a probability zero. However, this is obviously wrong. As one cannot cover all the possible n-grams which could appear in a language no matter how large the corpus is, and just because the n-gram didn't appear in a given corpus doesn't mean it would never appear in any text.

A common approach is then to generate a maximum-likelihood (ML) model for the entire collection and linearly interpolate it with a maximum-likelihood model for each document to create a *smoothed* model. We are not going into the details of smoothing methods here, but it suffices to say that *smoothing* is a technique to adjust the probability distribution over n-grams to make better estimates of sentence probabilities. A commonly adopted example is the Kneser-Ney Smoothing.

Here, we chose the similar but simpler scheme, named "Stupid Backoff", which was found to be less expensive to calculate while approaching the quality of Kneser-Ney Smoothing for significant amounts of data (Brants et al., 2007).

Then, if $w_1^L = (w_1, \ \dots, \ w_L)$ denotes a string of L words, such as "to be or not to be", over a fixed vocabulary and w_i^j any substring from the i-th to the j-th word of w_1^L, such as "or not to", the ML probability estimates for the n-grams are given by their (unnormalized) relative scores S as

$$S(w_i \mid w_{i-k+1}^{i-1}) = \begin{cases} \dfrac{f(w_{i-k+1}^{i})}{f(w_{i-k+1}^{i-1})} & \text{if } w_{i-k+1}^{i} \text{ was found in the corpus} \\ \alpha S(w_i \mid w_{i-k+2}^{i-1}) & \text{otherwise} \end{cases}$$

where the recursion above ends at unigrams $S(w_i) = f(w_i)/N$, with N being the size of the training corpus, and the value of backoff factor α, which may be made to depend on L, was empirically chosen to $\alpha = 0.4$ by Brants et al. (2007).

Therefore, in the example above, sticking to a 4-gram model,

$$S(be \mid or\ not\ to) = \begin{cases} \dfrac{f(\,or\ not\ to\ be\,)}{f(\,or\ not\ to\,)} & \text{if 'or not to be' was found in the corpus} \\ 0.4 \times S(be \mid not\ to) & \text{otherwise} \end{cases}$$

and down the recursion tree, if necessary.

Notice that the usual start-of-sentence marker $< S >$ is not needed here as the Stupid Backoff smoothing deals with unnormalized relative scores instead of probabilities.

Algorithm Rationale

Key considerations for algorithm design:

- A key point here is that the predictive model must be small enough to load onto the Shiny server. So pay attention to its size when creating and uploading it.

- When you type a phrase in the input box do you get a prediction of a single word after pressing submit and/or a suitable delay for the model to compute the answer?

Based on these requirements we chose a model based on existing 4-grams. For larger sentences, the user input is shortened to the last three words.

Besides, as, given a sentence s, we are interested only in the word w with the highest probability $P(w|s)$, the relative frequencies $\frac{f(w_{i-k+1}^{i})}{f(w_{i-k+1}^{i-1})}$ are previously computed and stored into the n-gram tables.

Therefore, at execution time, the Shiny app has only to look the 4-gram table for the desired s sentence and immediately recover the most probable next word w or, if necessary, go down the recursion tree along the 3-, 2-, and 1-gram tables. Again, as we are not interested in the probabilities themselves but only in the word with maximum-likelihood, the multiplications by the backoff factor α (Brants et al., 2007) are not really needed here.

Notice, however, that, despite the explicitly recursive expression above for $S(w_i|w_{i-k+1}^{i-1})$ involving $S(w_i|w_{i-k+2}^{i-1})$ in it, as R is not a recursive language, the actual implementation of this calculation had to be made bottom-up from unigrams ($S(w_i) = f(w_i)/N$) up as:

1. For every existing 1-gram w_i in the corresponding table, the corresponding $S(w_i)$ as $f(w_i)/N$ was calculated.
2. For every existing 2-gram in the corresponding table,

- it was decomposed as a 1-word sentence s followed by a w word,
- the corresponding $S(w|s)$ was calculated, and
- only the 2-gram with the highest $S(w|s)$ were kept for each distinct s.

3. Likewise, for every existing 3-gram and 4-gram in the corresponding tables.

The file **createTables.Rmd** shows how the raw data files are transformed into n-gram tables.

Notice also that, as the 4-grams, 3-grams, 2-grams, and unigrams have been all extracted from the same single corpus, the second case of the equation above, of w_{i-k+1}^{i} not being found in the corpus, simply cannot happen at this previous computation.

These tables (1-, 2-, 3-, and 4-grams) are loaded from compact .RData files before execution of the algorithm.

This choice was made to increase the speed of the algorithm and to reduce memory usage. It is possible that using a machine learning method with an unseen sequence of text would cost extra computational memory and time.

Repository Files

- **createTables.Rmd**: this file contains the code that takes the raw data and creates the 1-, 2-, 3-, and 4-gram probability tables .RData files, using the "Stupid Backoff" smoothing
- **profanitywords.RData**: a list of profanity words to remove used by the **prepareText.R** code, based on the list of profanity words obtained from the Shutterstock repo (http://github.com/shutterstock/List-of-Dirty-Naughty-Obscene-and-Otherwise-Bad-Words).
- **unigramFreq.RData**, etc.: the four 1-, 2-, 3-, and 4-gram probability tables files created by the **createTables.Rmd** code
- **global.R**: loads the lookup tables to search for matches, as well as additional libraries and functions
- **prepareText.R**: this is the main function called both by **createTables.Rmd** to clean the raw data and by **server.R** to clean the user input for the application interface
- **predictWord.R**: this is the function that predicts the next word and returns it to the user interface

- **server.R**: code necessary to access user input, calls functions necessary to clean the input, predict the next word, and return it to the user interface
- **ui.R**: code necessary for the application interface
    - Input: Text box that accepts a phrase
    - Output: The phrase as interpreted by the algorithm, and the predicted word

Shiny interface app

For the user interface, it was created a Shiny app with a simple and intuitive front end interface for the user. The text input is tokenized, its last 3 or 2 (or 1 if it's a unigram) words isolated and cross-checked against the tables to get the highest probability next word, which is then displayed.

The working app is available here (https://renatopdossantos.shinyapps.io/word_predictor/) and the slide deck is available here (http://rpubs.com/RenatoPdosSantos/WordPredictor).

References

- Brants, T., Popat, A. C., Xu, P., Och, F. J., & Dean, J. (2007). Large Language Models in Machine Translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, Prague, June 2007 (pp. 858-867). Stroudsburg, PA: ACL - Association for Computational Linguistics.
- Markov, A. A. (1906). Rasprostranenie zakona bol'shih chisel na velichiny, zavisyaschie drug ot druga (The extension of the law of large numbers onto quantities depending on each other). *Izvestiya Fiziko-Matematicheskogo Obschestva Pri Kazanskom Universitete*, 2-Ya Seriya, 15, 135-156.
- Peng, R. D. (2011). *Reproducible Research in Computational Science*. Victoria, CA-BC: Leanpub.
- Peng, R. D. (2016a). *Report Writing for Data Science in R*. Victoria, CA-BC: Leanpub.
- Peng, R. D. (2016b). *Exploratory Data Analysis with R*. Victoria, CA-BC: Leanpub.
- R Core Team (2016). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. Available at R site: https://www.R-project.org/ (https://www.R-project.org/).
- Toman, M., Tesar, R., & Jezek, K. (2006). Influence of Word Normalization on Text Classification. In *I International Conference on Multidisciplinary Information Sciences and Technologies (InSciT2006)*, Merida, Spain, October 25-28th, 2006 (pp. 354-358). Mérida: University of Extremadura and the Open Institute of Knowledge.