NLP: Smart Auto-Completion

|  |  |
| --- | --- |
| Dan Evgi  IDC  danevgi@gmail.com | Tom Huberman  IDC  hubermant@gmail.com |

|  |  |
| --- | --- |
|  |  |

Abstract

Is the knowledge about the context of a conversation gives better auto-completion? Can the knowledge of the recent used words improve the auto-completion? How good the trivial implementations are? In this document we will try to answer all this questions. To do so, we will present several implementations, and try to compare them with the trivial implementation by creating an evaluation system. We will show all the result and try to get a conclusion about all of our research questions.

1. Introduction

Auto-completion, or word completion is the process where a machine tries to predict what word is currently being typed or what will be the next word. Auto-completion mainly uses a dictionary of the words in the language and the probability of appearance; more sophisticated techniques are based on n-gram or part of speech tagging. The main use of auto-complete is to shorten typing time and avoiding spelling mistakes, and used in search bars, text message editors in cellular devises, source code editors, command-line interpreters and more.

In this study, we explore the addition of conversation context information to word completion, developing new algorithms in which the context of the last word that have been used will be considered in addition to the words themselves to improve the accuracy of the suggestions. We hypothesize that this will increase the likelihood of suggesting words that were recently added to our vocabulary in the current context, and enhance our prediction accuracy and will result better scores.

*He{y} Joe{} ho{w} {are} {you}*

1. General Instructions
2. Approach

Our approach was to start from several basic implementations, and then try to improve by using, knowledge of the conversation's context, knowledge of the last K-used words, n-gram tagging of the sentence, a smoothing mechanism for dealing with unknown words, user "erase" event, etc. What we did is to design a framework which allowing parsing a text file in a certain format, and apply several completion mechanisms which transfer the file in to a "CompledFile", which is a format we developed for represents the result. We designed the completion mechanism in a general way each word is separated in to letters which entered in to the completion mechanism, than the mechanism yields K proposals. This process continues until a word is completed correctly or if the word has ended. To be able to use all the different types of additional information, some events may be triggered during the completion process. Events can be anything, for example event represent a word that completed successfully or unsuccessfully, new sentence, the name of the writer (/sender), the user has erased a letter, etc. Each completer we develop decides whether or not to use an event's information.

The "CompletedFile" is in the following format, all punctuation marks are removed, and each word is written until the point where it was completed and the rest of the word is surrounded by curly brackets, in case of no completion the brackets are empty (figure 1).

* 1. Data and limitations

In our work we going to analyze the performance of several types of completion heuristics over several metrics that will introduced later, over IRC (Internet Reply Chat) conversations.

IRC is a protocol for live interactive internet text messaging widely used among open source communities around the world.

IRC format is ideal for our purposes since we wish to model the context influences about conversation auto completion and it contains extra metadata about each message that helps the process of context modeling.

Every message contains information about the time it sent, and the writer nickname, which supplies a way to distinguish the identities of the participants in the conversation and enables us to follow how the conversation is rolling.

Our data set contains more than 1GB of IRC text logs mainly from "Wikipedians" chat rooms and other communities, and covers a five year collection period.

1. Evaluation

Since this topic is examined mainly in the industry and not in research, there is no standard evaluation metrics, so we had to implement and invent our own, evaluation metrics. We use the following metrics:

* **POCW- Percentage Of Completed Words:** The number of words that achieved any completion divided by the total number of words in the test.
  + The higher the better. (between 0-1)
* **RSKR- Relative Saved keystrokes Ratio:** The ratio between the actual number of key strokes and the saved key strokes.
  + The lower the better. (positive number or infinity for no completion)
* **SKR- Saved keystrokes Ratio:** The ratio between the actual number of key strokes and the total number of key strokes needed.
  + The lower the better. (between 0-1)
* **CLPWS- Completed Letters Per Word Size:** The average, of the number of letters saved divided to the word length, per word.
  + The higher the better. (between 0-1)

The evaluation process is computed regard to the entire "CompletedFile", and regard to "CompletedFile" divided in to an equal sized group of sentences, in order to see the improvement of the completion process during time.

1. Implementation

Our project is divided in to three main *packages*, ***core***, ***eval*** and ***io***, all under the ***autocomplete*** package.

The ***core*** package contains all the classes connected to the completion process.

* ***autocomplete*.*core*.*completer-*** contains the implementations of all the type of completers.
* ***autocomplete*.*core*.*event-*** contains all the types of events that can be transferred to the completer.
* ***autocomplete*.*core.wordbank-*** contains the data structure containing the word statistics.

The ***eval*** package contains the python scripts that calculate the evaluation.

The ***io*** package contains all the readers and writers of the project. The reader that read and parse the given text, and the writers that writes the result in the "CompletedFile" format.

* 1. Base Line (N-Gram Completer)

The Base line completer is implemented in the ***BasicCompleter*** class. This type of completer, propose after each letter K completion proposals, which are the K most probable words to appear after the last N words.

* 1. Learning Completer

The learning completer is implemented in the ***LearningCompleter*** class. This type of completer, propose after each letter K completion proposals, which are the K most probable words to appear after the last N words. This completer is the same as the base line completer, accept it also listen to the ***SentenceEndEvent*** and each time it receives this event it update the probabilities by training on this sentence as well.

* 1. Context Completer

The context completer is implemented in the ***IRCCompleter*** class.

This completer saves knowledge about the n-grams frequency in the current conversation context aside the knowledge saved by the learning completer and ranks the words suggestions by combining both results.

The heuristic we used is to increase each word frequency of words in the current conversation context by a constant factor, and use the new frequencies to rank the next words suggestions.

When a word leaves the context, we decrease its frequency respectively in order to return its frequency to the frequency it would have if we hadn’t increased it artificially.

In our case we worked with IRC data, we defined the conversation context to be the current active IRC session.

There are other possible context definitions that we hadn't implemented such as the last-x-words, the last x sentences, the last x minutes (combined with time information from the IRC logs) and last x sentences per participant in the current session (combined with user information from the IRC logs).

All this possibilities shares the same outlines of our session implementation so we choose to omit them from our work.

* 1. Filter Completer

This type of completer is not stand for itself, rather a demonstration of concept that we found improves our results.

The filter completer is wrapper around any other completer described above, which remember the suggestions that have been already suggested to the user, and filter them out from the current word suggestion set.

For example if the user tries to enter the word **"Theatre"** and after two letters (**"Th"**) we suggests the word **"Theirs"** in the next letter **"e"** we won't suggest it again (even if its rank is higher than other suggestions) since the user already rejected it as the right completion.

This implementation is rather naïve for real usages since it assume that if the user rejects word in the first time it implies that this suggestion is not the right word, but it demonstrate the power of maintaining a state during completion of single word, and it may easily improve by using more complicate heuristics such as omit suggestions we suggested twice and more, or re-suggest suggestions we already rejected when the user enters a "delete" key.

1. Experiments
2. Results
3. Conclusions

References

Alfred. V. Aho and Jeffrey D. Ullman. 1972. The Theory of Parsing, Translation and Compiling, volume 1. Prentice-Hall, Englewood Cliffs, NJ.

American Psychological Association. 1983. Publications Manual. American Psychological Association, Washington, DC.

Association for Computing Machinery. 1983. Computing Reviews, 24(11):503-512.

Ashok K. Chandra, Dexter C. Kozen, and Larry J.Stockmeyer. 1981. Alternation. Journal of the Association for Computing Machinery, 28(1):114-133.

Dan Gusfield. 1997. Algorithms on Strings, Trees and Sequences. Cambridge University Press, Cambridge, UK.