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.

figure 1

***/bin> java -cp ../lib;. autocomplete.core.Main <parameters>***

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.

figure 2

***/src> python autocomplete/eavl/eval.py <file-to-evaluate>***

1. General Instructions
   1. Testing completers

In order to run our experiments you have two options, run our JUnit tests, or use the Main class made for testing specific completer.

In order to run the Main class you'll have to supply the following arguments, number of suggestions, n-gram, test files, results file and training files. The test file and train files should be IRC formatted log files, and the number of suggestions represents the size of suggestion set which suggested for every word in the test file.

Run the Main class can be done from the */bin* folder in the project source, by the following command:

* 1. Evaluating results

In order to evaluate completion result file, you may execute our evaluation script located in *src/autocomplete/eval/eval.py*. The script takes only the file to evaluate as an argument, and prints the result to screen, and to *res.xlsx* file which created in the current working directory (creating the excel results format requiring *pandas* package to be installed).

To run the evaluation script you may use the following command:

1. 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 3).

* 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 evolves.

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.

figure

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

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
   1. Down the rabbit hole

At first we didn't had all the IRC data, so our first experiment was against *Alice's Adventures in Wonderland* children book. The rationale behind this decision was that in a book the plot evolves much like in a conversation and new characters places and terms appears as it evolves.

This experiment showed us the importance of the completer's adaptiveness, as we saw in the *"Caterpillar"* word, which appears at the middle of the story, and stays until the end of it. *"Caterpillar"* is a unique word, which unlikely to appear in previous train data. When we first run a naïve completer this word was never completed successfully, and being relatively long and frequent word in Alice lower the completer performance. Adding simple learning heuristic that after each sentence fish learn its words, improved the completer performance significantly.

* 1. Completer comparison

We compared the four completers we implemented by testing them twice on two IRC files, on long IRC file (1081kb) with 6 files of train data, and on short IRC file (306kb) with 3 files of train data.

In each of the tests we used same model parameters which were 1 gram and 3 suggestions per step. For the context completer we used 4 as the context factor.

The purpose of the experiment was to determine if our complete heuristics are indeed improve the results and if so how much.

* 1. Find optimal N-Gram Size

In order to find the optimal n-gram size for the best completer from the previous experiment we run that completer with same number of suggestions and the different n-grams values.

The purpose of the experiment was to determine the best value of n-gram to continue with.

* 1. Influence of suggestion number

In this experiment we knew that if we increase the size of the suggestion set we would get better evolution results. This experiment purpose was to find out how much increasing the suggestion set size actually improves the completer score.

The experiment was made by running the same completer with same parameters and same training data on the same test file (short test file), with different size of suggestion set.

* 1. Influence of training size

Since we had a lot of training data, we wanted to find out how much more training really influence our completer scores. In this experiment we run the same completer with the same environment (parameter + test file) with different size of training data.

* 1. Influence of context factor

This experiment purpose was to find out what is the optimal value of the context factor parameter.

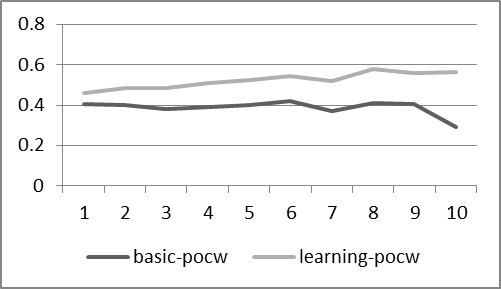
As mentioned in section 5.3, the context factor determine how much the completer would be sensitive regard to the context content, were 0 indicates no context influence exactly like in regular learning completer, and higher context factor indicate higher sensitivity to the context, masking previous knowledge from the training data.

1. Results
   1. Down the rabbit hole



figure

We can clearly see that the learning completer scores much better results than the baseline (see that *Caterpillar Effect* from section 6.1).



figure

We can see that there was an improvement of more than 11% in the percentage of completed words in the text just by learning words that already appeared.

The above graph shows the improvement of the **pocw** metric over segments of 60 sentences. As we can see after few segments the difference between the learning and the basic completers becoming significant.

* 1. Completer comparison



figure

As we can see in the table above the results are inconclusive. On one hand it seems that the context heuristic improves the **pocw** metric, which as mentioned means that we complete more word successfully which is a positive finding, but on the other hand it seems that the pure learning heuristic scores better on the **skr** metric which means that in total it saves more keystrokes. Even though we tested on different test files, with different amount of train data, we can see that the results are persistent among themselves.

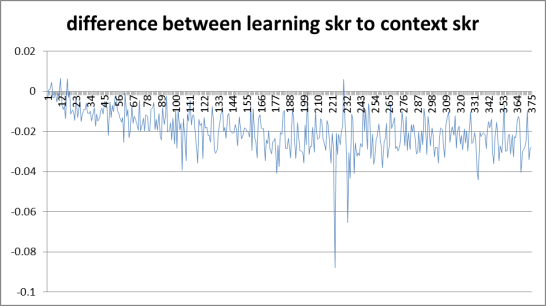
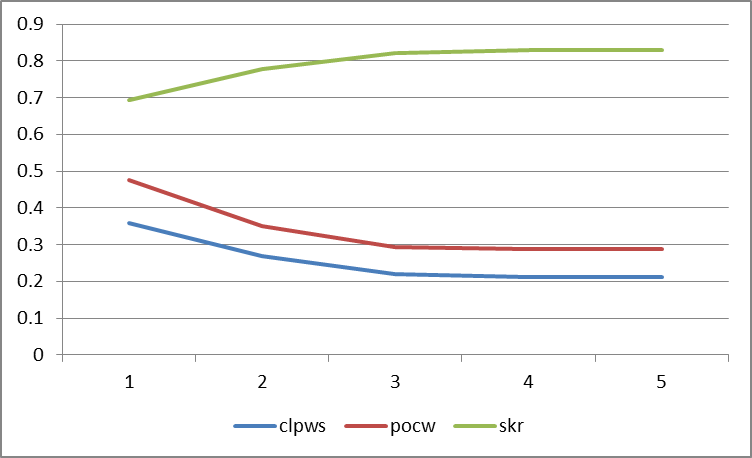
We relate the value differences between the results of the two tests to the differences in the amount of train data we used in each test. Since the **skr** difference is minor between the learning completer and the context completer we continue to test the filtered context completer in the rest of work.

figure 7

* 1. Find the optimal N-gram size



figure



figure

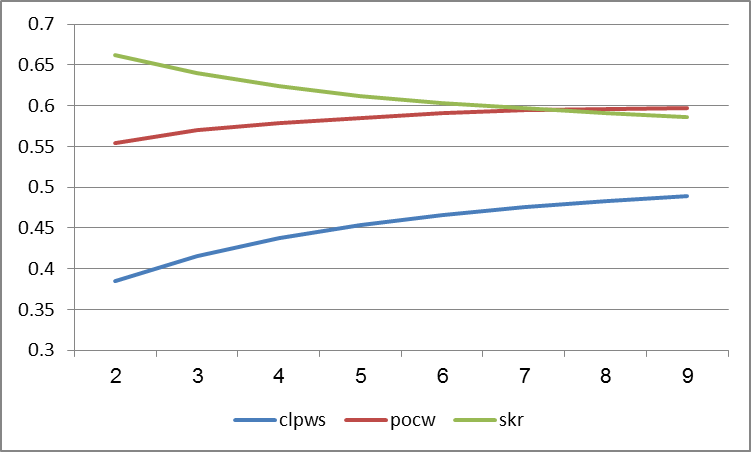
As easily reflected from the above results the completer scores the bests results with unigrams.

This rather reasonable as we preform those tests with the same train data (which were about 10kb), which probably wasn't enough for training higher n-gram values.

* 1. Influence of suggestion number



figure 10



As we can see, and as was expected increasing the size of suggestion set, improve the performance of the completer. But also as we can see the improvement seems to be bounded and from certain size this improvement is insignificant.

figure 11

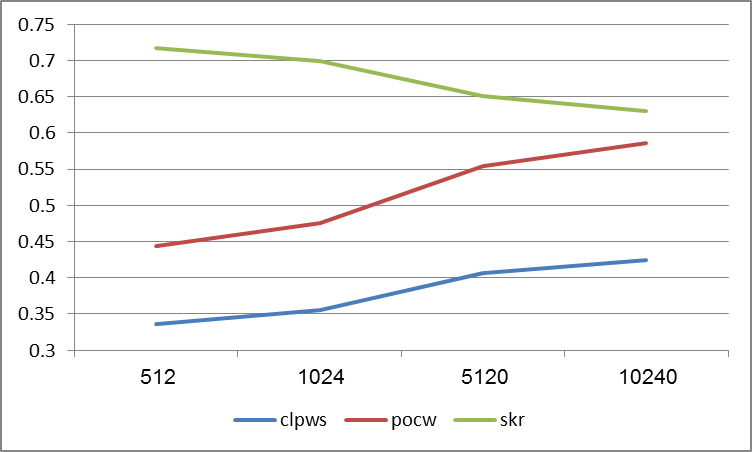
Increment the size of suggestion set, even though improves performance, is most of the time impossible though to both UI limitations (small display) and since it might confuse the user of the system.

In the rest of the work we decide the keep with 3 suggestions.

* 1. Influence of training size



figure



figure

As we can see, and as expected, when the completer is trained on a large amount of data, its results improves. The improvement, though seems significant from the results, would have eventually and inevitably become less and less significant and we believe that if we would have continue with this direction we would have found it limit.

* 1. Influence of context factor



figure

The context factor changes have barely affected the evaluation metrics, as we can see from the above table. The explanation for this phenomenon is that the main improvement is made from the concept of context, and the context factor is less sensitive parameter for this range of values.

We assume that for lower context factor values we would have got slightly more interesting results.

* 1. Results locations

All the results locate under the result folder in the project. Each result is formed from three files, *.txt* file which contains the completer results, *.eval* file which contains text summary of the evaluation metrics, and *.xlsx* excel file which contains the same information as the *.eval* file in more continent way.

* Section 7.1 results are:
  + res-alice-basic
  + res-alice-learning
* Section 7.2 results are:
  + res-irc-baseline-1gram-test-GA.2008715
  + res-irc-learning-1gram-test-GA.2008715
  + res-irc-test-GA.2008715
  + res-irc-with-filter-1gram-test-GA.2008715
  + res-irc-baseline-1gram-test-short
  + res-irc-learning-1gram-test-short
  + res-irc-test-short
  + res-irc-with-filter-1gram-test-short
* Section 7.3 results are:
  + res-irc-with-filter-\*gram-sfactor4
* Section 7.4 results are:
  + res-irc-with-filter-1gram-sfactor4-\*suggestions
* Section 7.5 results are:
  + res-irc-with-filter-big-train(\* bytes)
* Section 7.6 results are:
  + res-irc-with-filter-1gram-sfactor\*-3suggestions

1. Conclusions and future work

Our final results about context base autocomplete are inconclusive, and it's hard to determine whether or not the consideration of the context helps for the autocomplete process from our results. Nevertheless, our results focused about specific set of data, and specific context definition (IRC session), which was might be misled direction.

Despite that, we developed a set of tools to create and evaluate autocomplete heuristics, which may enable easy way for farther research in this field, and may the foundation for future works. With this framework, new heuristics or new definition of context can be explored (see section 5.3).

Another possible contributes from our research for farther work in this field is the enormous collection of data we gathered that is now available for all.

References

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