A language independent user adaptable approach for word auto-completion

Abstract—**In this paper, we address the problem of word auto-completion for free text (e.g. messages, emails, articles, poems, etc.) written in different languages. We focus on improving the user experience by developing a user-oriented model that is able to learn different writing styles, while still providing initial predictions without any user written documents. We show that by learning from the user, the performance of an auto-completion system can be improved by up to 18 %. In order to keep query processing times low, we deploy a binary search technique that retrieves groups of words from an inverted index based on their first letters. This retrieval method reduces the query processing time by up to 80%.**

I. INTRODUCTION

With the increasing usage of mobile devices, and devices with limited typing facilities, it is highly desirable to have solutions to speed up typing. The goal of such a system is to predict words (and phrases) while the user is typing, thus allowing for faster writing and increasing productivity.

The auto-completion problem is not new. Such solutions have been used for years in a variety of activities. The most common of these activities is everyday text writing tasks. Nowadays almost everyone owns a smartphone device. Most of these devices have a built-in software that suggests words while the user writes messages, emails, etc. Another use case is query predictions in search engines. For instance, Google (and others) display a list of suggestions when someone starts typing in the search field. Other applications of auto-completion systems include command line suggestions, code completions in IDEs, etc.

The general flow of an auto-completion system is: (1) A query is made to the system (for example the first letters of a command or file in the command line), (2) the system processes the query and (3) returns suggestions.

A query for word auto-completion usually consists of 2 or 3 previous words, and the first letters of the desired word. For example, consider typing the following text: *'I am go'*. At this point, a word completion query would be triggered with the previous words being *['i', 'am']* and the first letters being *'go'*. Hopefully the system answers with *'going'* or *'goofy'*.

The component of an auto-completion system that usually stores possible completions is the model.

II. RELATED WORK

Hogler Bast and Ingmar Weber [4] discuss the possibility of using an inverted index [6] for answering queries made to search engines. They present the basic principle of using a normal inverted index, and the retrieval function of word based on their common documents. Furthermore, they also introduce an improved inverted index that enhances the runtime on query processing. Their inverted index structure reduces query processing time ten times compared to the one of compressed state of the art indexes.

Milad Shokouhi [5] presents a possible supervised ranking framework for learning user search preferences based on the user's long term search history and location. They prove that personalized rankers improve the performance of regular popularity based rankers by 9%.

An important note is that for an autocompletion system to be useful, it must be able to process queries fast. Research [2], [3] shows that the upper time limit for to the human eye to perceive instantly is around 100 ms.

Moreover, one can also push the limits further, and make whole phrase predictions. Arnab Nandi and H. V. Jagadish [1] propose an approach for phrase auto-completion that uses a data model similar to a pruned count suffix tree (PCST) [8]. They solve two problems that occur with phrase auto-completion: efficiently storing phrases and how much of a phrase to predict. For this they introduce the Fuzzy Tree data structure that stores phrase words similar to how a suffix tree stores word prefixes. They also define the notion of a significant phrase and use this to determine how much of the phrase to predict.

III. CONTRIBUTIONS

A. Motivation

Some autocompletion systems suggest words based on their occurrence frequencies alone. As each user has a personal writing style, it is highly desirable that an autocompletion system learns these styles. This ensures that the system adapts to the user, and is able to make predictions accordingly. However, the system should have the ability to predict in a cold start context (new user scenario).

B. Conceptual Design

Ideally, the designed system tackles both word and phrase auto-completion. This requires a clear separation of the query triggers for each part. An obvious solution is to trigger word completion queries with each alphabetic character, and phrase completion queries with each space character. Moreover, the system must work on multiple languages at once. To do this, the design must be loosely coupled, in order to change components with ease. This allows it to load different data models and switch to the corresponding query processors during runtime. Figure 1 presents a possible component diagram.

Formally, a text editor extension listens for input characters. It switches between the two auto-completion processors depending on the input. Once a processor is chosen, a query

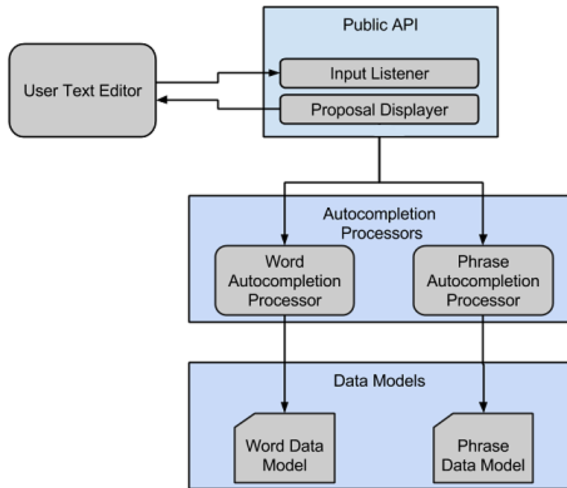


Fig. 1. The component diagram of a word and phrase autocompletion system

is issued to it and the result is returned in the form of a list, regardless of which processor was used to answer the query.

C. Proposed Approach

Our approach aims to be a language-independent one. We focus on improving user experience by developing an user-oriented model that is able to learn different writing styles, while still providing initial predictions without any user written documents. Our approach employs a hybrid strategy starting from the probabilistic and inverted index models [6]. We take into account the one dimensional positions of words in the documents, and use these to define the context in which a word is used in relation to previous words.

For this, we first propose an extension to the inverted index model, the User Oriented Index, that stores word positions in documents and marks words that appear in user written documents. The User Oriented Index is discussed in section IV of this paper. We illustrate what changes from the simple inverted index through an example.

Secondly, we introduce a fast binary search retrieval based on the first word letter (Bidirectional Group Boundary Identification), and a post-processing ranking phase based on (1) word occurrence frequency in general documents, (2) word occurrence frequency in user documents and (3) the one dimensional distance between words and the previous words from the query. Both of these are presented with more details in section V of this paper.

We finally evaluated and compared our solution with a simple index that ranks solely on frequencies. We use three data sets that have different sizes, topics and languages (two of them are written in English, and one in Romanian).

IV. Data Model

The section starts with a short exemplification of the inverted index and then enhances it by including information about the user and word positions in documents.

TABLE I: Example of Inverted Index pruned with OCC_TH = 2

Word	Posting List	Word	Posting List
i	[doc1, doc3]	*is*	[doc2]
the	[doc1, doc2, doc3]	*filled*	[doc2]
market	[doc1, doc2, doc3]	*with*	[doc2]
people	[doc2, doc3]	*am*	[doc1]
going	[doc1]	*hate*	[doc3]
to	[doc1]	*it*	[doc3]
when	[doc3]	*fill*	[doc3]

A. The Inverted Index

A widely used data model for word auto-completion is the Inverted Index [6]. This is a dictionary of terms (sometimes also referred to as a vocabulary lexicon) that stores, for each term, a list of documents the term occurs in. Each item in the list - which records that a term appeared in a document (and possibly the positions in the document) - is conventionally called a posting (the list is called postings list).

To show the inverted index in auto-completion solutions, consider the following documents:

1) doc1: ”I am going to the market”
2) doc2: ”The market is filled with people”
3) doc3: ”I hate it when people fill the market”

The resulting inverted index is presented in table I.

We experimentally found that an uniform pruning a common [9] task in the context - with an occurrence threshold (OCC_TH) that accounts for document length (like the one in equation (1)) performs good.

$$OCC_TH = 5 * 10^{-6} * nChars \quad (1)$$

In equation (1) nChars is the number of characters in the documents. Nandi and Jagadish use this idea of varying threshold as well [1].

For the purpose of this example, lets consider a occurrence threshold of 2. All the words from our inverted index that have frequencies less than OCC_TH have been marked with italics. Pruning reduces the size of the index, which also decreases the overall runtime and increases the performance by ensuring that only the most frequent words are considered as valid suggestions.

An auto-completion query made to an Inverted Index is composed of a set of previous words (*PW*) and the first n letters of the desired word (*FL*). The result of such a query is a list of words *[w1, w2, w3, ...]* where *w1, w2, ...* all start with the letters *FL*, and appear in the same documents as the previous words *PW*. Then, a post-processing ranking step sorts the returned words *[w1, w2, w3, ...]* based on some criteria.

Let’s now consider that someone types the following: *”People are in the mark...”*. A query of the form *Q* = { *PW: [’in’, ’the’], FL: ’mark’* } is triggered. The system searches for all the words that start with *’mark’* and have common documents with either of the words *’in’* or *’the’*. This results in the word *’market’* being returned as a completion proposal.

B. User-Oriented Index

In order to be able to learn from the user, and adapt to his/her writting style, we developed the User-Oriented Index which, besides storing words and posting lists like the simple inverted index, marks those words that come from user documents, and records information about user specific word positions and frequencies. Nevertheless, we still desire initial predictions, without having user written documents. We do not want to have two different models for initial and user predictions, as this adds extra logic to the system, and initial words may still be of interest after having user-written document. In order to achieve our goals, we decided to separate the two concepts (initial prediction and user prediction) at document level:

- Initial documents: those documents that make the initial index (the default model that comes with the system). These document should be spread on a large variety of topics, and contain a rich vocabulary.
- User documents: those documents that are written by the user. These are used to update the initial index as the user writes them, and words that come from these documents are given a higher priority at ranking.

We differentiate between the two document types by using a *UserDocumentMask*. This is a mask that gives user written documents higher document ids. For example, using a *UserDocumentMask* of 100, regular documents get ids between [0, 100], and user documents from 101 to infinity. For our tests, we used a mask of 10000.

A more permissive occurrence threshold should be used for user documents. Let's call this the user occurrence threshold (USER_OCC_TH on short). The user occurrence threshold in equation (2) allows ten times more user words in the index compared to the one (1).

$$USER_OCC_TH = 0.5 * 10^{-6} * nChars \tag{2}$$

Using this extra user information allows us to rank by considering all of the following:

- word frequency in initial documents
- word frequency in user-written documents
- the one dimensional distance between the word and the previous words from the query in the common documents.

This requires a more complex ranking algorithm, which is discussed in the section V of the paper.

As an example, assume that the initial index is the pruned index from table I and we have the user written document: *"Today I was at the market"*. The following assumptions are made:

- the general OCC_TH is 2 (same as before)
- the USER_OCC_TH is 0, s.t. all user words are inserted in the index.
- the index is pruned, s.t. no words with frequencies less than OCC_TH appear in it.

TABLE II: Example of User-Oriented Index pruned with OCC_TH = 2 and USER_OCC_TH = 0

Word	Posting List
i	1 : [1], 3 : [1], 101:[2]
the	1 : [5], 2 : [1], 3 : [7], 101 : [5]
market	1 : [6], 2 : [2], 3 : [8], 101 : [6]
people	2 : [6], 3 : [5]
today	101 : [1]
was	101 : [3]
at	101 : [4]

- document ids are used (i.e. instead of *doc1, doc2, ...* we use *1, 2, ...*) with an *UserDocumentMask* of 100.

The resulting User-Oriented Index is presented in table II.

Query processing works the same as for the simple Inverted Index. The difference between the systems is at the ranking level.

C. Building the Index

Building the index involves a two step procedure.

1) Pre-process step to convert all text to lowercase, split the text in words and remove any word that contains non-alphabetical characters (a-z).
2) Indexing step to record for each word output in step 1 the document ID and position in the document.

The algorithm for reading a file and appending its contents in the index is presented in algorithm 2. The procedure is split in two separate functions. The *preprocess* function 1 cleans the raw contents from the file, and remove any words that are contain non-alphabetical characters. The second function, *appendToIndex* 2, reads the contents of the file, calls the *preprocess* function in order to clean the contents, and then appends all the words to the index using an *update* function.

Algorithm 1 Preprocess build step

```
1: function PREPROCESS(rawContent)
2:     processed ← []
3:     for all word in rawContent.split() do
4:         if word.isAlpha() then
5:             processed.append(word.lower())
       return processed
```

Algorithm 2 Index construction algorithm

```
1: function APPENDTOINDEX(index, filePath)
2:     rawContet ← read(filePath)
3:     preprocessed ← PREPROCESS(rawContent)
4:     isUserDoc ← CHECKUSERDOC(filePath)
5:     docId ← GETLASTDOCID(index)
6:     if isUserDoc then
7:         docId ← docId + userDocumentMask
8:     docPosition ← 0
9:     for all word in preprocessed do
10:        UPDATE(index, word, docId, docPosition)
11:        docPosition ← docPosition + 1
       return processed
```

D. Storage and Encoding

Encoding (or compression) [7] is a commonly used method to reduce the size of a data model. Although in general smaller data models are preferred, we did not use any compression or encoding for our index. The reason for this is that we want to focus on having small run times for queries, therefore the whole model should be loaded into the main memory, so minimizing the storage size is not that important. The important thing is to have a small uncompressed index size, without introducing an additional decompression step that takes CPU cycles. This is ensured by the varying occurrence threshold. Nevertheless, we tried different encoding techniques like Variable Byte Encoding and Run Length Encoding, but due to the structure of our model, these are only able to encode word position lists as document IDs are keys in a dictionary. A comparison between the techniques is presented in table V of the Experiment and Results section.

V. Query Processing and Ranking

A. Word Retrieval

A query to the system consists of a set of previous words and the first letters of the desired word. After receiving a query like this, we first search for all the words that start with the given letters. To keep runtimes as low as possible, we deploy a binary search technique on the words in the index (for this, the index needs to be sorted!). An illustration of this algorithm (Bidirectional Group Boundary Identification, or *BGBI* on short) can be seen in figure 2. In this example we search for all entries that start with *ab*.

The BGBI algorithm employs the following strategy:

1) find **any** word that starts with the given group of letters (*FL*) using binary search.
2) create two position sentinels:
   a) one of them decreases until the word on the current position no longer matches the FL group (st)
   b) the other increases until the word on the current position no longer matches the FL group (end)
3) return all words with positions in the range created by the two sentinels.

We called it the Bidirectional Group Boundary Identification (BGBI) because it identifies the boundaries for the group of words that start with the same letters by searching in two directions (bidirectional).

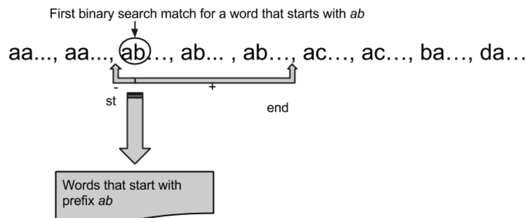


Fig. 2. Illustration of BGBI algorithm

This algorithm keeps the query processing time small even for huge data sets due to the search space reduction. On average, there is a 80% reduction in execution time when using this algorithm compared to a linear filtering algorithm. See more results in the section VI of this paper.

B. Ranking

The next step is to rank the retrieved words. Our ranking approach considers both the initila and use. In order to do so, our ranking algorithm computes the score as the average distance between the wanted word positions and the positions of the previous words (*freqScore*), multiplied by the influence of user documents. We defined this influence in terms of a *userInfluence* variable and the word's frequency in user documents (*uOcc*). Thus the resulting score is computed by equations 3 and 4:

$$freqScore(w, PW) = \frac{\sum_{docId} dist(w, PW, docId)}{freq(w)} \tag{3}$$

$$score(w, PW) = freqScore(w, PW) * userInfluence^{uOcc} \tag{4}$$

Because we compute our scores based on the distance between words, smaller scores are considered better. As we want a word that appears in user documents to get a better score, we have to minimize it. We do this by giving values between 0 and 1 to the *userInfluence* variable, and applying equation (3) to the general score *freqScore*.

In practice there are situations when no words that match the first letters have any common documents with the previous words. In this case we propose that any words that match the letters are returned, and these are ranked using a naive frequency based ranking.

VI. Experiments and Results

This section starts by presenting two metrics for computing the word distances in documents (*the dist() ranking function*), followed by a comparison between utilization of the Simple vs. User Oriented index models. In the entire section, we are mainly interested in measuring the precision, recall and runtimes obtained by the different solutions. For computing precision and recall, we use the rank-based metrics 5 and 6, introduced by Nandi et al. [1]:

$$RankPrecision = \frac{\sum 1/rank(accepted_autocompletion)}{n(predicted_autocompletion)} \tag{5}$$

$$RankRecall = \frac{\sum 1/rank(accepted_autocompletion)}{n(queries)} \tag{6}$$

A. The dist() ranking function

We discuss different approaches to computing the distance between two words in a document. We implemented two variations:

- Simple Linear Distance: computes the simple linear distance between the words. i.e. returns *abs(pos(w1) - pos(w2))*

- Gaussian Distance using Normal Distribution (ND): computes the Gaussian distance between the two words using the ND function.

We argue that the Gaussian distance should perform better, as there are cases in which a word that is in the middle of the previous words gets better ranking than one that is immediately after one of them. For example, consider the scenario in figures 3a and 3b. The possible completion word positions are marked with * and the positions of previous words represent the mean of the ND functions. It is clear that the second word (the one right after the blue ND) should be preferred, as it is directly after a previous word.

The linear distance clearly favors the word between the NDs, as the other one has a much bigger distance to the left-most ND mean. In the case of the Gaussian distance, the left-most ND has no influence on either words, as both of them are outside its area. Thus only the right-most ND function is used to compute the score, and this favors the second word.

In our experiments, we observed that the distances between positions of words queried for auto-completion are large (around 500 word positions). This makes the Gaussian distance impractical, as the standard deviation of the ND function has to be greater than *0.25*DocLength* (we deduced this from experiments) in order to be able to rank the desired words correctly (i.e. the ND has to spread on over a quarter of the document length). This gives the same results as the linear distance, as most words are influenced by all the NDs.

A short comparison in terms of precision, recall and runtime for the two methods is presented in table III.

TABLE III: Comparison of distance function implementations

Distance Function	Precision	Recall	Runtime
Gaussian	77%	71%	8 ms
Linear	77%	71%	3 ms

We therefore decided to use the linear distance, as the results are the same and there are less computations made, thus saving some CPU cycles.

B. Simple Index vs. User Oriented Index

In order to properly compare the two data models, we conducted our experiments on three data sets.

The first one is a small data set consisting of blog and news articles together with user-written Facebook messages in the Romanian language (further denoted by *SmallRo*). This set contains 72,000 words, and takes 3 MB of disk size. We used a web crawler to gather the blog and news articles. The Facebook messages belong to one of us.

The second data set is a bit bigger, and it consists of some random Wikipedia articles, and user written documents on Software Products in the English language (further denoted by *MediumEn*). This set has 1 million words and takes 6 MB of disk space. We downloaded the Wikipedia articles, and the documents for Software Products are written by one of us.

The third data set is large, and consists of documents written in English, on different topics: Woodworking, Fitness, Cycling, Computer games, Gadgets, Phone and Notebook reviews, Chemistry, Math, Web technologies, Economy, Travel, Food recipes, etc. This set spreads on over 7.4 million words, and has a size of 46 MB. We obtained the documents from different web sources using a web crawler. For this data set, we consider that a possible user has documents written on the following topics: Chemistry, Food recipes and about traveling in India. We shall refer to this data set as *BigEn*.

We ran all experiments on an Intel i7 dual core, 3.7 MHz processor with 8 GB RAM memory, using the Python programming language. After building the data models, we stored them in json format, without any compression or encoding. One can also use a low-cost database like MongoDB to store the models, but we found it easier to keep them in a simple text file.

We built our User-Oriented Index, considering that all user document from our data sets are written by one user, and the Simple Index from all the data sets, and compared the building times, size on disk and the number of words and documents in the index. The build results are shown in table IV. We use the previously discussed occurrence thresholds for our builds (OCC_TH = (1) and USER_OCC_TH = (2)).

The varying threshold becomes much higher with larger data sets (e.g. 186 for the large English data set), thus less words occur in the index. It can be seen that for smaller sets the threshold is more permissive. Also, the fact that we use a different threshold for user documents allows more words into the User Oriented index. We set the value of our user influence variable to *userInfluence = 0.2*

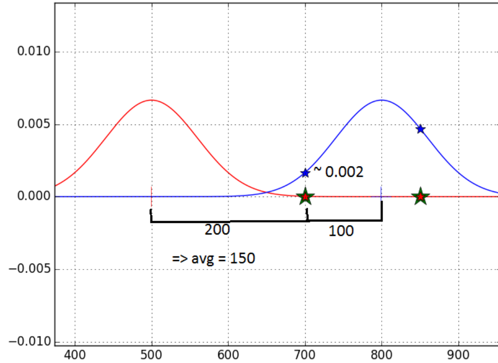
TABLE IV: Comparing build measurements for the Simple Index and the User Oriented Index

Data Set	Model	Build Time	Size	nWords	nDocs
SmallRo	User Oriented	9 s	4.3 MB	22205	120
	Simple	6 s	1 MB	19166	120
MediumEn	User Oriented	8 s	6.5 MB	4011	230
	Simple	6 s	1 MB	2276	230
BigEn	User Oriented	82 s	48 MB	16512	3550
	Simple	138 s	7 MB	2183	3550

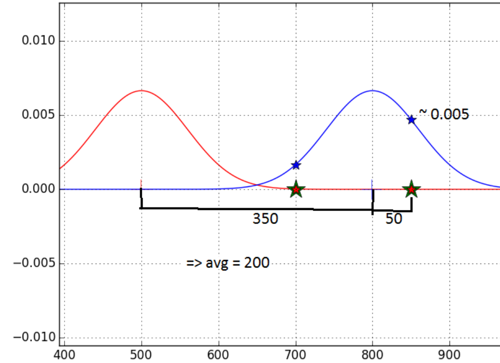
As expected, our User-Oriented index grows in size due to the extra information about word positions it stores. A method to avoid this is splitting the index into smaller indexes based one some features like the topic of documents(as we attempt to do in our future work). Having the intention to reduce this size, we compared different encoding techniques like Variable Byte and Run Length Encoding for the large English data set. The results are presented in table V.

TABLE V: Comparison of different encoding techniques

Measurement	RLE	VBE	No Encoding
Size on disk	42 MB	42 MB	48 MB
Query Runtime	50 ms	120 ms	9 ms



(a) First Word



(b) Second Word

Fig. 3. Example of Gaussian Distance vs Linear Distance: First Word represents measurements of both distances for the first possible completion word, while Second Word for the second possible completion word

The small size reduction is justified by the fact that we cannot encode document ids, only word position lists. If we would modify the model to create different lists for document ids and position lists, then it might be possible to have better size reduction. i.e., an entry in the index would look like this: *market:{docs: [1, 2, 3, 101], poss: [[6], [2], [8], [6]}* where the first position list in the poss collection corresponds to the first document in the docs collection. Although this is a direction to be investigate, we focus for now on obtaining high precision and small runtimes, and not using any encoding seems to be the best choice.

We build both our User-Oriented Index and the simple Inverted Index for all datasets, and test each model. Test documents represent documents from Facebook messages for the SmallRo data set, Software Products for MediumEn, and Food recipes for the BigEn. We consider all test documents as user-written. We did not use any encoding at this step.

Although we generally consider a word to be predicted correctly if it appears in the first three positions of the resulted list, we still prefer that it appears on the first position, thus penalizing the second and third positions using the above metrics.

In order to test the system, we pass over the test documents with sliding window of 3. The first two words represent the previous words of the query, and the third one is the desired word. We consider the first 4 letters of the desired word for querying, and then check which (if any) of the words in the resulted list correspond to the desired word.

The results are presented in table VI.

TABLE VI: Precision, Recall, RT comparisons for the two models

Data Set	Model	Precision	Recall	Runtime
SW Products	User Oriented	89%	87%	1
	Simple	71%	61%	1
Facebook Messages	User Oriented	80%	78%	4
	Simple	71%	68%	4
Food recipes	User Oriented	84%	82%	6
	Simple	76%	66%	6

As it can be seen, the User Oriented model behaves better in all of our tests, outperforming the simple model with 18% on the Software Products dataset and 8% on the big dataset with user documents on Food recipes. The interpretation is that a user-specific solution, that favors user-written words, is able to rank them with higher precision

Another important aspect to follow is the speed of learning users behavior. To estimate this, we plotted learning curves of retrieval performance against the amount of indexed words for the *BigEn* data set (Precision: figure 4, Recall: figure 5) and the *MediumEn* data set (Precision: figure 6, Recall: figure 7).

We observed that both the User Oriented and Simple Index systems learn when content that is relevant to the tests (e.g. content from the same discussion) comes in. The difference is that the User Oriented system has a bigger learning step. For example, consider the Software Products data set in figure 6. One can see that around 3000 words relevant content is indexed. Our User Oriented system boosts up its precision with around 20%, while the Simple Index system only with 13%.

The reason why the learning plots achieve better results compared to the ones in the tables is that the OCC_TH is more permissive. i.e. we prune based on how much we learned until now, while when building the whole model at once we prune based on the whole data-set size.

We also compared the runtime of the Bidirectional Group Boundary Identification (BGBI) algorithm with that of a linear word retrieval that uses a filter, by plotting them as a function of the number of words indexed so far. This graph is shown in

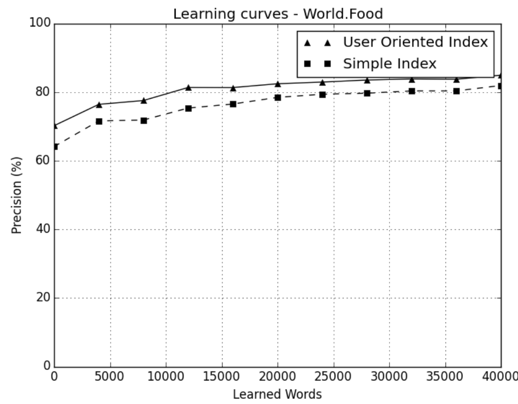


Fig. 4. Precision comparison between Simple Index and User Oriented Index on BigEn

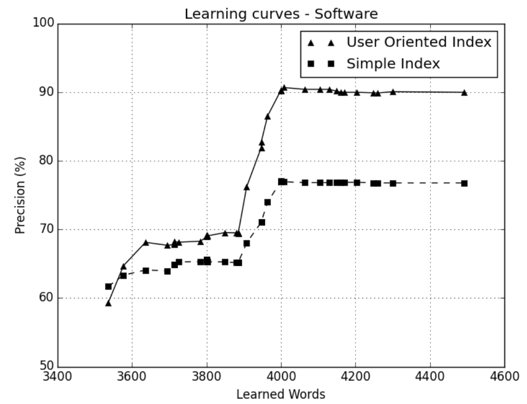


Fig. 6. Precision comparison between Simple Index and User Oriented Index on MediumEn

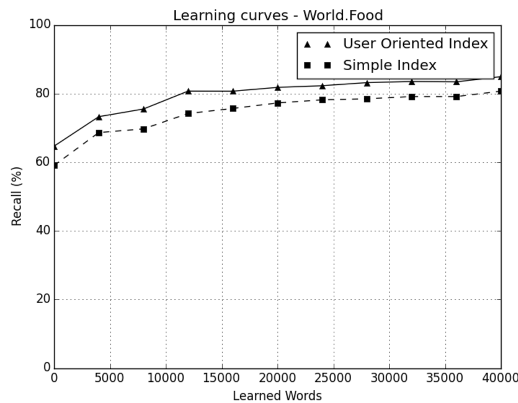


Fig. 5. Recall comparison between Simple Index and User Oriented Index on BigEn

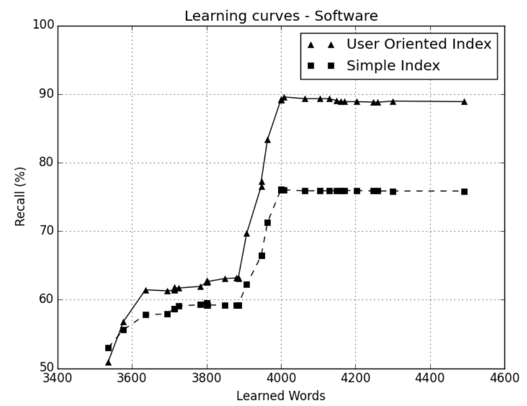


Fig. 7. Recall comparison between Simple Index and User Oriented Index on MediumEn

figure 8 below. One can see how the BGBI runtime increases much slower compared to the linear retrieval runtime (this actually goes over 200 ms). It is important to note that the varying OCC_TH assures that the index size stays small, and this results in smaller runtimes as well.

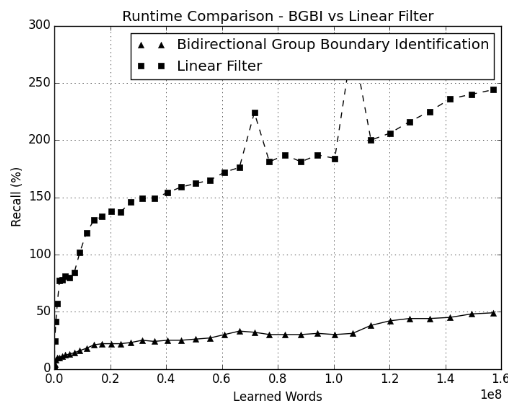


Fig. 8. Comparison of the BGBI algorithm and a Linear Filter

VII. Conclusion

In this paper, we introduced an extension to the commonly used inverted index, called the User Oriented Index, which stores user information at document level by using id masks. Together with a new ranking strategy that makes use of this information, our solution improves the learning capabilities of an auto-completion system by up to 35%, and increases the performance with 18%. Finally, we present a word retrieval technique, called the Bidirectional Group Boundary Identification (BGBI), that is based on the Binary Search algorithm. By using the BGBI algorithm, we reduced the query processing times by up to 80% compared to a linear retrieval that uses a filter function.

References

[1] Arnab Nandi and H. V. Jagadish: Effective Phrase Prediction. VLDB '07 Proceedings of the 33rd international conference on Very large data bases Pages 219-230

[2] S. Card, G. Robertson, and J. Mackinlay. The information visualizer, an information workspace. Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology, pages 181186, 1991.

[3] R. Miller. Response time in man-computer conversational transactions. Proceedings of the AFIPS Fall Joint Computer Conference, 33:267277, 1968.

[4] H. Bast and I. Weber: Type Less, Find More: Fast Autocompletion Search with a Succinct Index. SIGIR '06 Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval

[5] Milad Shokouhi: Learning to personalize query auto-completion, Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval Pages 103-112

[6] Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schtze. Introduction to information retrieval. Vol. 1. Cambridge: Cambridge university press, 2008.

[7] : Trotman, Andrew. "Compressing inverted files." Information Retrieval 6.1 (2003): 5-19.

[8] P. Krishnan, J. Vitter, and B. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. Proceedings of the 1996 ACM SIGMOD international conference on Management of data, pages 282293, 1996.

[9] Carmel, David, et al. "Static index pruning for information retrieval systems." Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval. ACM, 2001.