Topic Oriented Auto-completion Models
Approaches Towards Fastening Auto-completion Systems

Stefan Prisca[1], Mihaela Dinsoreanu[2] and Camelia Lemnaru[3]
[1]*Computer Science, Technical University of Cluj-Napoca, Vitei 7, 551107, Medias, Romania*
[2]*Computer Science, Technical University of Cluj-Napoca, Baritiu Str. 26-28, RO-400068, Cluj-Napoca, Romania*
[3]*Computer Science Department, Technical University of Cluj-Napoca, 26 Baritiu St., room c9, 400027, Cluj-Napoca,*

Keywords: Word Auto-completion, Topic Oriented Data Models, Topic Indexing.

Abstract: In this paper we propose an autocompletion approach suitable for mobile devices that aims to reduce the overall data model size and to speed up query processing while not employing any language specific processing. The approach relies on topic information from input documents to split the data models based on topics and index them in a way that allows fast identification through their corresponding topic. Doing so, the size of the data model used for prediction is decreased to almost one fifth of the size of a model that contains all topics, and the query processing becomes two times faster, while maintaining the same precision obtained by employing a model that contains all topics.

1 INTRODUCTION

With the increasing usage of mobile devices, and devices with limited typing facilities, it is highly desirable to have solutions that 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 represented by query predictions in search engines. For instance, Google (and other engines) 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. Another important usage of auto-completion is in systems that assist persons with writing disabilities in performing daily task on a computer.

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 prefix being 'go'. Hopefully the system answers with 'going' or 'goofy'. The component of an auto-completion system which stores possible completions is the model.

Usually, there is one model for an auto-completion system, which leads to the following problem: in data sets which spread on multiple topics, there is influence induced from one topic to the others. This means that words/phrases from one topic will be mixed with words from the others. For example, if a data set contains documents from totally different topics like automobile constructions and software development, there are scenarios in which the auto-completion system might propose *automobile* instead of *algorithm*.

2 RELATED WORK

There has been a lot of work in the field of auto-completion, with most recent focus on query auto-completion for search engines.

The authors of (Bast '06) present a simple auto-completion system, that makes use of an inverted in-

Prisca, S., Dinsoreanu, M. and Lemnaru, C..
Topic Oriented Auto-completion Models - Approaches Towards Fastening Auto-completion Systems.
In *Proceedings of the 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2015) - Volume 1: KDIR*, pages 241-248
ISBN: 978-989-758-158-8

dex (Manning '08). Although this is not the main purpose of their paper, they present the basic principle of using a normal inverted index, and the word retrieval function based on the documents they commonly appear in. The approach is relatively simple, the hash-like structure of the inverted index allows for fast word retrievals, as it is proven by (Bast '06), and there is room for a lot of extensions over the inverted index.

Such an extension to the inverted index is discussed by the authors of (Prisca '15). Here, an user oriented alternative to the inverted index data model is presented, called User Oriented Index. This new model is based on the idea that in an auto-completion system there are two document types: (1) initial documents, those documents that are used for general index construction, and (2) user documents: documents that are written by the user, and that are given higher priority. They specifically separate user documents in order to give higher ranking to words that come from these documents. To achieve this, the User Oriented Index makes use of document id masks to identify user-written documents. These masks are used to split the document id domain in two: (1) initial documents $[0 \rightarrow userDocMask)$ and (2) user documents $[userDocMask \rightarrow \infty]$. Moreover, in order to improve final ranking precision, this new index keeps track of word positions from the input documents. The results showed an average improvement of 13% over the regular inverted index, in terms of prediction quality and learning capabilities. On the other hand, storing all that additional information requires more space, and the User Oriented Index takes approximately 1.3 times more size compared to a regular index. This approach requires a good size reduction strategy that will not influence the performance.

More recent research has been done in the area of query auto-completion for search engines. Taking another user-oriented approach, Milad Shokouhi (Shokouhi '13) 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%. Although this system improves the performance of regular rankers, it is a supervised learning method that requires the model to be learned.

All of the non-learning approaches discussed rely on a single, big model to make predictions. This model contains all the topics from the data set, mixed together, which may result in predicting words from the wrong topics. Also, this leads to large model sizes, and requires more and more pruning, encoding, etc. Moreover, another fact to keep in mind is that for an auto-completion system to be useful, it must be able to process queries fast. Research (Card '91), (Miller '68) shows that the upper time limit for to the human eye to perceive instantly is around 100 ms. This is a big constraint, as the predictions must appear immediately after a key is pressed, otherwise risking to be outdated. This again is hard when having only one huge model to predict completions against, as when its size grows, so does the query processing time. One alternative to consider is defining a strategy that separates this one big model into smaller ones that are easier to store and provide faster query processing while offering the same performance.

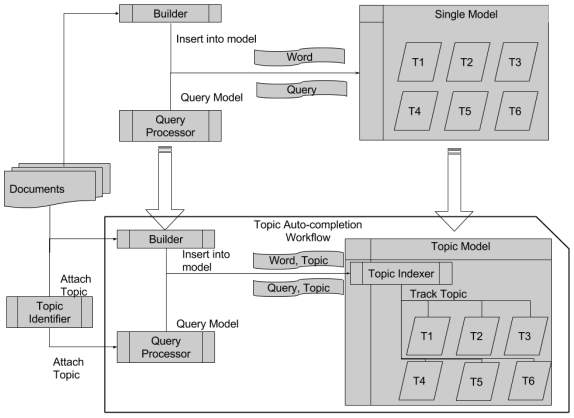

Figure 1: Main Topic Oriented model building and query processing workflow.

3 PROPOSED APPROACH

In this paper we will present a solution to the one model problems, that aims to reduce the size of the model, reduce the query processing time and still offer the same completion capabilities as standard models (even better in some cases, as shown by our results). Our approach leverages topic information to split the model based on the topics contained inside it, and load the corresponding sub-model when making query completions.

Figure 1 illustrates how the main building and querying workflow is altered to include additional logic to identify and attach topic information, and to track the topics. Instead of communicating directly with the model, the Builder and Query Processor pass their messages to a Topic Indexer. The job of the Topic Indexer is to track topics inside the model, and provide the required information to the builder and query processor. We can identify the two workflows:

1. *Builder Workflow*: The builder takes in documents and inserts them into the corresponding model by

passing the *Word* and *Topic* information into the topic indexer. For our purposes, we assume the topic is known.

2. *Query Processing Workflow* The query processor sends and auto-completion query to the Topic Indexer, and attaches the additional *Topic* information. Again, we assume that the topic is known.

The main constraint of such an approach is that the system needs enough input data in order to be able to find the topic, and only then make predictions. Therefor we consider as input data long documents (e.g. e-mails, essays, reports, etc.), not short in-line messages. We also assume that our topics are already identified for each of the documents, thus the problem of topic identification is not covered within this paper. The main focus remains on presenting solutions for the Topic Model and Indexer, and how the Builder and Query Processor communicate with them. The part of the Topic Model Overview diagram 1 that we cover in this paper is the *Topic Autocompletion Workflow*.

The issues with one model solutions that we identified are (1) topic interferences, (2) big models and (3) slow query processing. In order to address these issues, we present the following methods of storing topic information:

- have one model for all topics, and include a list of topics for each gram. (one-for-all) Although this method does not directly address problems 2 and 3, it aims to reduce the topic interferences, but not totally separate the model, as words from different topics are mostly shared between the topics. The obvious disadvantage is that there still is a single model that stores all words, regardless of their topic. The advantage is that each word is stored once, even if this word appears in multiple topics.
- have a separate model for each topic. (one-for-each) This method provides a more extreme solution to all of the issues above. It separates the models in completely different sub-models, and predicts queries using only the sub-model corresponding to the topic at hand. The disadvantage is that words are shared between multiple topics. If most of the words in the data-set are shared between topics, this method will store all of the shared words in each sub-module, thus resulting in extra copies of the same word which will ultimately increase the overall size required to store all of the sub-models.

In order to avoid confusion, in the following sections we refer to the overall data model as super-model, and to separate topic-specific models as sub-models. Moreover, throughout this paper, we consider that each document has one topic. In cases when a document has multiple topics, we consider only the most relevant one.

3.1 One-For-All Topic Model

The first idea that comes in mind is to create a super-model that stores topic information for each gram. The problem that remains after is how to store this information inside the super-model? One can insert for each gram a list of topics it appears in. For example, in the case of an inverted index data model, a list of topics in which the word appears is stored together with the postings list. An entry from the User Oriented index containing this extra information looks like this: `market:{postings: {1 : [1], 3 : [1], 101:[2]}, topics:{ food, automobile, shopping }}`.

The obvious issue with this solution is the size. The super-model will grow a lot due to topic information. Imagine a word like *how*. As it most probably appears in all topics, it has a large topic list that affects the running performance of the system. To address this problem, we propose to generalize the idea concerning user document ids presented in (Prisca '15), by applying it to topics. This results is a new Topic Model: the Topic Oriented Index.

To achieve this, we define a mapping between each topic identified in the data set, and a range of document ids. For this mapping, any common data structure can be used, but it is important for it to be a singleton. This means that once topics are mapped to a certain id range, the same mapping is used throughout the whole system. Moreover, since we cannot always have all topics identified, there needs to be an id range dedicated to uncategorized documents. We refer to this as the *Uncategorized* id range, and propose that it takes values at the end of the id domain (i.e. $[lastKnownTopicId \rightarrow \infty)$, where *lastKnownTopicId* is the last id assigned to a known topic). In case new topics appear in the data set, one only has to reduce the set of *Uncategorized* id ranges, and allocate some of them to the new topic.

As an example, let us consider that we have a data set consisting of the following documents, split by topic:

- *doc1, Topic:Architecture*: This building has an old Gothic facade design, made by a famous architect
- *doc2, Topic:Software*: Our system architect chose the Facade design for this particular problem
- *doc3, Topic:Mathematics*: The problem can be represented in an equation system.

The document topics from above are: *Architecture*, *Software*, *Mathematics*. As there are few documents in each topic, we shall use id ranges of *100* for this example, and assign them to each topic: (1) *Architecture*: $[0 \rightarrow 100]$, (2) *Software* : $[101 \rightarrow 200]$, (3)*Mathematics*: $[201 \rightarrow 300]$ and *Uncategorized*: $[301 \rightarrow \infty)$. The simple Inverted Index that would result from these documents is presented in 1 and the Topic Oriented Index in table 2. Both the indexes have been pruned with an occurrence threshold (OCC_TH) of 2.

Table 1: Example of Inverted Index pruned with OCC_TH = 2.

Word	Posting List
facade	{0, 1}
the	{1, 2}
this	{0, 1}
design	{0, 1}
an	{0, 2}
architect	{0, 1}
system	{1, 2}

Table 2: Example of Topic-Oriented Index pruned with OCC_TH = 2.

Word	Posting List
facade	{0 : [7], 101 : [6]}
the	{101 : [5], 201 : [1]}
this	{0 : [1], 101 : [9] }
design	{0 : [8], 101 : [7]}
an	{0 : [4], 201 : [7]}
architect	{0 : [13], 101 : [3] }
system	{101 : [2], 201 : [9]}

As a comparison to the regular Inverted index, the Topic Oriented Index respects the entry format of the User Oriented Index ((Prisca '15)), which means that it uses the document id masks, and stores word positions from the documents. Furthermore, we want to store user information as it is done in the User Oriented Index. The same idea with a user mask can be applied. This means that the document id will be composed from a user id, a topic id and the actual document id within the topic range. Formally, the document id can be defined as $< userDocMask > + < topicIdRange >$. Considering the topics from our previous example, if a user with $userDocMask = 1000$ wrote documents about mathematics, then the range of ids associated to those documents is $1000 + [201 \rightarrow 300] = [1201 \rightarrow 1300]$.

Using this kind of document ids allows us to store information about multiple users in the same index by having more *userDocMask*s. Continuing our previous example, we can associate a *userDocMask* of 2000 to a second user. Assuming this second user writes about software programming, we now encode his/her documents with ids in range $[2101 \rightarrow 2200]$.

3.1.1 Building the Topic Oriented Index

The Builder Workflow of the Topic Oriented Index involves the same basic strategy as building the User Oriented Index (Prisca '15). The only difference is that for the Topic Oriented Index, the logic for document id assignment changes. We now need to store the last used id for each topic, and append the user mask to that id. The build procedure is presented in algorithm 1. This algorithm implements the Builder Workflow for the Topic Oriented Index.

Algorithm 1: Topic Oriented Index builder workflow algorithm.

```
1: topicMapping ← LOADTOPICMAPPINGS()
2: function APPENDTOINDEX(index, filePath)
3:     rawContet ← read(filePath)
4:     docTopic ← IDENTIFYTOPIC(rawContent)
5:     preprocessed ← PREPROCESS(rawContent)
6:     docId ← GETLASTDOCID(topicMapping, docTopic) + 1
7:     docPosition ← 0
8:     for all word in preprocessed do
9:         UPDATE(index, word, docId, docPosition)
10:        docPosition ← docPosition + 1
11:    UPDATELASTDOCID(topicMapping, docId)        return
   processed
```

3.1.2 Query Processing for the Topic Oriented Index

Query processing and ranking workflow operations are similar to the corresponding operations in the User Oriented Index. The difference for the Topic Oriented Index is that we can no longer compare the document id to a single mask, but instead we have to compare it to the topic id interval. Formally, we can present this approach as algorithm 2, which implements the Query Processing workflow for the Topic Oriented Index.

Algorithm 2: Topic Oriented Index Query Processing.

```
1: KnownTopic ← GETCURRENTTOPIC()
2: function TOPICINDEXQUERYPROCESSOR(query)
3:     if KnownTopic == Null then
4:         idRange ← [0 → ∞]
5:     else
6:         idRange ← GETIDRANGE(KnownTopic)
       return PROCESSQUERY(query, idRange)
```

3.2 One-For-Each Topic Model

The second approach to having topic information, is to split the super-model in totally separate sub-models for each topic. This kind of model separation clearly

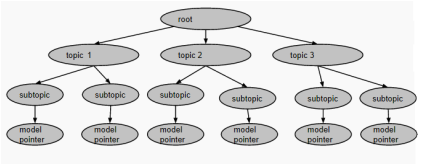

Figure 2: One-For-Each Topic Model.

distributes sub-models among all topics. This reduces the size of the used model, as each of the sub-models will only contain a subset of the grams contained in the super-model. One could use various data structures to map each of the topics to the corresponding sub-model, like a table, a list of tuples, a tree, etc. We have decided to use a tree structure for our purposes, as it allows us to easily track the topic hierarchy (the Topic Tree). This is illustrated in figure 2.

The One-For-Each topic model relies on having an underlying auto-completion system for each subtopic, which has all the auto-completion logic implemented. This means that the mere job of the One-For-Each topic model is to simply track the topics, and delegate any auto-completion work to the underlying system. It can be thought of as an extension for existing auto-completion models, that adds the logic to track topics.

3.2.1 Building Topic Tree Models

As mentioned, the Builder Workflow for the Topic Tree Model will identify the topic, and delegate all other work to the underlying auto-completion system. The algorithm for building topic models in presented bellow. 3.

Algorithm 3: Topic Tree Model building algorithm.

```
1: function BUILDTOPICMODELS(rawInput)
2:     topicPath ← getTopicPath(rawInput)   ▷ Retrieve the full topic path of the given input
3:     wordModel ← BUILDINDEXMODEL(rawInput)
4:     SAVEWORDBYTOPIC(wordModel, topicPath)
5:     UPDATETOPICINDEX(topicPath, wordModel)
```

3.2.2 Query Processing on Topic Tree Models

An issue with total topic separation arises at query processing: what happens when we do not know the topic of the current document (i.e. the document for which the system currently processes queries)? The system will not be able to make predictions, as models for different topics are disjoint. In order to overcome this, we have to combine the topic oriented auto-completion strategy with a single model auto-completion strategy. The solution is to first use a single model in order to make general predictions. After the user writes enough content to allow a topic identification system to determine the topic, we switch to the sub-model corresponding to that topic, and use that to make predictions. The algorithm for query processing is presented in 4. This is just an adapter to the underlying model's query processor. The only purpose of this algorithm is to load the proper model into memory. This algorithm implements the Query Processing workflow for the Topic Tree Models.

Algorithm 4: Topic Tree Model Query Processing.

```
1: KnownTopic ← Null
2: UsedModel ← SingleModelPath
3: function TOPICQUERYPROCESSOR(query)
4:     if KnownTopic == Null then
5:         if UsedModel ≠ SingleModelPath then
6:             UsedModel ← LOADMODEL(singleModelPath)
7:     else
8:         modelPath ← GETMODELPATH(KnownTopic)
9:         if UsedModel ≠ modelPath then
10:            UsedModel ← LOADMODEL(modelPath)
       return PROCESSQUERY(query, UsedModel)
```

4 EXPERIMENTS AND RESULTS

In this section we present the results obtained by topic oriented data models, and compare them to single models. We are mainly interested in measuring the precision, recall and runtimes obtained by using various data models for word prediction systems (note that we do not experiment with phrase predictions). The precision and recall are computed using the Mean Reciprocal Rank, as presented in equations 1, (2).

As data models for comparison, we chose two that are topic oriented, and two that are not:

1. single, non-topic User Oriented Index: the user focused auto-completion data structure presented in paper (Prisca '15) To separate between general documents and user documents, we use a *userDocumentMask* of 30000 ((Prisca '15)).
2. Topic Oriented Index: The altered Inverted Index data structure presented in this paper. This data structure encodes topic information using document ids. For building the Topic Index, we consider topic id ranges of 1000, and an *userDocumentMask* of 30000. This means that the first topic from our data set will get ids $[0 \rightarrow 1000] \cup [30000 \rightarrow 31000]$, the second will have $[1001 \rightarrow 2000] \cup [31001 \rightarrow 32000]$, and so on. The second interval of id ranges corresponds to user documents on the given topic. We assume that in our data set there are not enough documents and top-

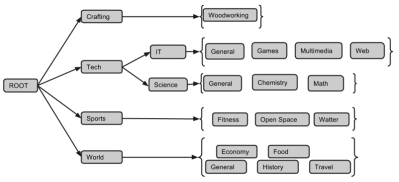

Figure 3: Topic structure of the BigEn data set.

ics to consume all the ids allocated within the intervals.

3. Topic Tree with an underlying User Oriented Index: The tree-like indexing structure for topics that uses underlying User Oriented Indexes for word storage. This data structure was presented in this paper. For the underlying User Oriented Index, we used the same *userDocumentMask* of 30000.
4. single, non-topic regular Inverted Index: the regular Inverted Index data structure, commonly used for information retrieval tasks. A short introduction to this data structure is given by (Bast '06) and (Manning '08).

In order to test the topic models, we used data sets that spread on multiple topics. We marked the topic of each document by hand. Note that we do not use a topic identification system to find the topic of a document. In a real-life situation, the topic identification system might introduce additional errors. Because our system is dependent on the topic, for our experiments we consider having large documents, like e-mails, essays, articles, software documentations, etc. This ensures that the topic for the document at hand is known.

We conducted tests on a large data set, consisting 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. For the purpose of our experiments, we make the assumption that only one user has documents in our model, and all the user-written documents are written by the same user. This set is referred to as BigEn for the rest of this paper. This data set contains documents that spread over the topic hierarchy presented in figure 3.

The experiments have been conducted 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 build a Topic Index, a Topic Tree, a single Simple Index and a single User-Oriented Index. We use the whole data set for all models, such that all of them are built from the same user and nonuser documents. After the build, a pruning step is applied, with an occurrence threshold of $5 * 10^{-6} * nChars$ for initial general documents, and of $0.5 * 10^{-6} * nChars$ for user documents. The results obtained after building the models are presented in table 3. We compared the size on disk, build time, number of words and number of documents that result in each model. Since the topic tree involves multiple separate models, we present the average statistics.

Table 3: Comparing build measurements of our models.

Model	**Build Time**	**Size**	**nWords**	**nDocs**
Topic Tree	5.6 s	10 MB	5000(avg)	97(avg)
Topic Index	88 s	51 MB	16512	3550
User Oriented Index	82 s	48 MB	16512	3550
Simple Index	138 s	7 MB	2183	3550

The User Oriented Index and Topic Index both have the same number of words in them. The reason is the the Topic Index uses a underlying User Oriented Index to build its data model, therefor the resulting words are the same. The difference between the two is that the Topic Index takes more space due to having larger document ids from topic ranges. The Topic Tree takes less average size, has less average documents and has the fastest build. Although the Topic Tree contains on average 1/3 of the words in other models, these do not spread on that many documents, thus having a more reduced size (1/5 of the User Oriented Index).

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 as the letters passed to the query, and then check which (if any) of the words in the resulted list correspond to the desired word. For the topic based models, we switch between different indexes based on the current topic. For testing, we assumed that a single user has documents written on the following topics *Chemistry*, *Food recipes*, *India*. The

test results are presented in tables 4,5,6, corresponding to each of the user topics.

To measure the performance of the systems, we use the Mean Reciprocal Rank metric for precision 1 and recall 2. 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 lowering the scores of the second and third positions by using the above metrics.

$$RankPrecision = \frac{\sum 1/rank(accepted\_autocompletion)}{n(predicted\_autocompletion)} \quad (1)$$

$$RankRecall = \frac{\sum 1/rank(accepted\_autocompletion)}{n(queries)} \quad (2)$$

It can be seen that the User-Oriented Index, the Topic Oriented Tree and the Topic Oriented Index perform mostly the same. Although there are not large variations among the three, the Topic Oriented Tree tends to outperform the other two, due to the fact that it contains words from one topic only, and it totally eliminates interferences from other topics. This also improves its runtime. As it can be seen, it's half the runtime of other systems. Nevertheless, all three solutions outperform a simple index in terms of precision and recall.

Table 4: Test results of topic based vs single models on Chemistry documents.

Model	Precision	Recall	Runtime
Topic Index	73.48%	72.83%	6 ms
Topic Tree	72.95%	72.23%	14 ms
User Oriented Index	73.03%	72.30%	14 ms
Simple Index	63.74%	59.93%	12 ms

Table 5: Test results of topic based vs single models on Food recipes documents.

Model	Precision	Recall	Runtime
Topic Index	83.39%	81.83%	8 ms
Topic Tree	88.04%	88.04%	4 ms
User Oriented Index	84.09%	82.51%	8 ms
Simple Index	77.16%	67.34%	8 ms

We are also interested in how fast our solutions are able to learn from the user. To find out, we plotted learning curves for the BigEn data set. We compared the learning capabilities of the User Oriented Index, the Topic Oriented Tree, and the Topic Oriented Index. We create two testing scenarios:

Table 6: Test results of topic based vs single models on documents about traveling in India.

Model	Precision	Recall	Runtime
Topic Index	75.94%	72.15%	9 ms
Topic Tree	79.56%	78.35%	5 ms
User Oriented Index	76.38%	72.56%	9 ms
Simple Index	65.82%	51.31%	10 ms

- *scenario one*: how fast the systems learn with no other user-written documents except the ones they learn from
- *scenario two*: how the systems learn when there are user-written documents in the data set on other topics

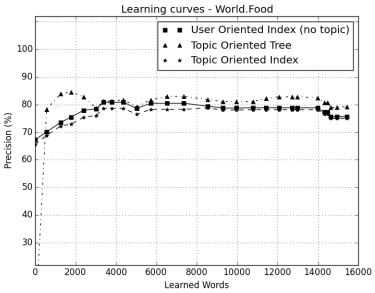

Figure 4: Learning comparisons - scenario one.

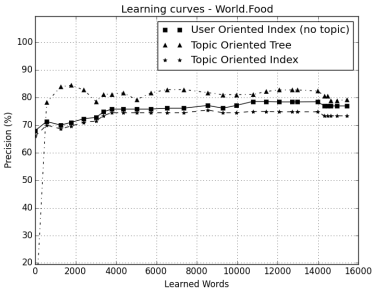

Figure 5: Learning comparisons - scenario two.

During our learning tests, we observe that the Topic Oriented Tree tends to learn much faster, but it is also more sensitive. The interpretation is that it contains few words at the beginning, and whatever is indexed can improve or decrease its precision. It can be seen in both plots that around 2000 words, the learning curve of the Topic Oriented Tree drops a little. However, it adapts faster than the other two. After 500 words its precision increases to over 80%, while the other two are around 70%. To be noted that the learning curve of the Topic Oriented Tree does not change between the two test scenarios. Due to the fact that this model separates topics completely, it is irrelevant if there are other topics used by the user, as

these will have other data models. On the other hand, the Topic Oriented and Simple Index both suffer from topic interferences. With no other user topics, these two reach 78% (4), but when there are other topics (5) they stagnate at 70%.

5 CONCLUSION

In this paper, we presented prediction data models that are split by document topics, and which achieve the same results as regular models, while having less then 25% of the size (on average) and requiring half the query processing time. We conclude that in order to benefit from the advantages of both single models and topic split models, one can build single models at first, and as the data set grows, start using topic oriented data structures. This keeps runtimes and sizes small even after learning from a large collection of documents, and ensures that the system can make prediction even when the topic is unknown, by using the single model.

REFERENCES

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.

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

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.

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

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.

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.

Stefan Prisca, Mihaela Dinsoreanu, Rodica Potolea. "A language independent user adaptable approach for word auto-completion", 11th International Conference on Intelligent Computer Communication and Processing, 2015.

Jiang, Jyun-Yu, Yen-Yu Ke, Pao-Yu Chien, and Pu-Jen Cheng. "Learning user reformulation behavior for query auto-completion.", In Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval, pages. 445-454. ACM, 2014.

Whiting, Stewart, and Joemon M. Jose. "Recent and robust query auto-completion.", In Proceedings of the 23rd international conference on World wide web, pp. 971-982. ACM, 2014.

Milad Shokouhi: Learning to personalize query auto-completion, Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval, 2013, Pages 103-112