Effective Phrase Prediction

Arnab Nandi
Dept. of EECS
University of Michigan, Ann Arbor
arnab@umich.edu

H. V. Jagadish
Dept. of EECS
University of Michigan, Ann Arbor
jag@umich.edu

ABSTRACT

Autocompletion is a widely deployed facility in systems that require user input. Having the system complete a partially typed "word" can save user time and effort.

In this paper, we study the problem of autocompletion not just at the level of a single "word", but at the level of a multi-word "phrase". There are two main challenges: one is that the number of phrases (both the number possible and the number actually observed in a corpus) is combinatorially larger than the number of words; the second is that a "phrase", unlike a "word", does not have a well-defined boundary, so that the autocompletion system has to decide not just what to predict, but also how far.

We introduce a *FussyTree* structure to address the first challenge and the concept of a *significant* phrase to address the second. We develop a probabilistically driven multiple completion choice model, and exploit features such as frequency distributions to improve the quality of our suffix completions. We experimentally demonstrate the practicability and value of our technique for an email composition application and show that we can save approximately a fifth of the keystrokes typed.

1. INTRODUCTION

Text-input interfaces have been undergoing a sea change in the last few years. The concept of automatic completion, or *autocompletion* has become increasingly pervasive. An autocompletion mechanism unobtrusively prompts the user with a set of suggestions, each of which is a suffix, or completion, of the user's current input. This allows the user to avoid unnecessary typing, hence saving not just time but also user cognitive burden. Autocompletion is finding applications in specialized input fields such as file location fields [30], email address fields [30] and URL address bars [25], as well as new, more aggressive applications such as dictionary-based word / phrase completion [23] and search query suggestion, available now in mainstream web browsers [25]. With the recent increase in user-interface to server communication paradigms such as AJAX [32], and as users become more receptive to the idea of autocompletion, it will find many more applications, giving rise to increased expectations from autocompletion systems.

Current autocompletion systems are typically restricted to *single-word* completion. While being a very helpful feature, single word completion does not take advantage of collocations in natural language – humans have a tendency to reuse groups of words or "phrases" to express meanings beyond the simple sum of the parts. From our observations, examples of such collocations can not only be proper noun phrases; such as *"Enron Incorporated"*, but also commonly used phrases such as *"can you please send me the"* or *"please let me know if you have any questions"*. Such phrases become much more common when attached to a personalized context such as email, or personal writing, due to the tendency of an individual to mention the same the same phrases during repetitive communication. Many current systems incorporate multiple word occurrences by encoding them as single words; for example *"New York"* is converted to *"New_York"*. However, such schemes do not scale beyond a few words, and would bloat the index (e.g. suffix-tree) sizes for applications where we desire to index a large number of phrases. It is this multi-word autocompletion problem that we seek to address in this paper.

We note that autocompletion is only useful if it appears "instantaneous" in the human timescale, which has been observed [5, 27] to be a time upper bound of approximately 100ms. This poses a stringent time constraint on our problem. For single word completion, typical techniques involve building a dictionary of all words and possibly coding this as a trie (or suffix-tree), with each node representing one character and each root-leaf path depicting a word (or a suffix). Such techniques cannot be used directly in the multi-word case because one cannot construct a finite dictionary of multi-word phrases. Even if we limit the length of phrases we will consider to a few words at most, we still need an "alphabet" comprising all possible words, and the size of the dictionary is several orders of magnitude larger than for the single word case.

It goes without saying that autocompletion is useful only when suggestions offered are correct (in that they are selected by the user). Offering inappropriate suggestions is worse than offering no suggestions at all, distracting the user, and increasing user anxiety [1]. As we move from word completion to phrase completion, we find that this correct-

Supported in part by NSF grant number 0438909 and NIH grant number R01 LM008106.

VLDB 2007 Vienna, Austria

ness requirement becomes considerably more challenging. Given the first 5 letters of a long word, for example, it is often possible to predict which word it is: in contrast, given the first 5 words in a sentence, it is usually very hard to predict the rest of the sentence. We therefore define our problem as one of completing not the full sentence, but rather a multi-word *phrase*, going forward as many words as we can with reasonable certainty. Note that our use of the word "phrase" does not refer to a construct from English grammar, but rather an arbitrary sequence of words. At any point in the input, there is no absolute definition of the end of the current phrase (unlike word or sentence, which have well-specified endings) – rather the phrase is as long as we are able to predict, and determining the length of the phrase itself becomes a problem to address. For example, given the prefix *please let*, one possible phrase completion is *please let me know*; a longer one is *please let me know if you have any problems*. The former is more likely to be a correct prediction, but the latter is more valuable if we get it correct (in that more user keystrokes are saved). Choosing between these, and other such options (such as *please let me know when*), is one problem we address in this paper.

Traditional methods of autocompletion have provided only a single completion per query. Current autocompletion paradigms and user interface mechanisms do allow for multiple suggestions to be given to the user for example by cyclic tab-based completions [39] in command line shells, or by drop-down completions in search engines. However, this facility is typically used to present a comprehensive list of possible completions without any notion of ranking or summarization.

There are dozens of applications that could benefit from multi-word autocompletion, and the techniques we develop in this paper should be applicable irrespective of the application context. Nonetheless, to keep matters concrete, we will focus on email composition as our motivating application. We all spend a significant fraction of our work day composing emails. A tool can become valuable even if it helps decrease this time only slightly. We hypothesize that the typical official email conversation is rerepetitive and has many standard phrases. For each user, there is typically available a long archive of sent mail, which we can use to train an autocompletion system. And finally, the high speed at which most of us type out emails makes it essential that autocompletion queries have very short response times if they are to help rather than hinder the user, leading to tight performance requirements which we will attempt to address.

1.1 Our contributions

We introduce a query model that takes into account the multiplicity of completion choice, and propose a way to provide *top-k, ranked suggestions*, paying attention to the nature of results returned.

We introduce the concept of a *significant phrase*, which is used to demarcate frequent phrase boundaries. It is possible for significant phrases to overlap. It is also possible for there to be two (or more) significant phrases of differing lengths for any completion point. To evaluate systems that provide multiple ranked autocompletion results, we define a novel total profit metric for ranked autocompletion results, that takes into account the cost of distraction due to incorrect suggestions.

At the physical level, we propose the construction of a word-based suffix tree structure we call the *FussyTree*, where every node in the tree corresponds to a word instance in the corpus, and root-leaf paths depict phrases. Queries are executed upon receipt of a prefix phrase from the user-interface, and the suffix-tree is then traversed to provide the results. We develop techniques to minimize resource requirements in this context, such as tree size and construction time.

The next section discusses the challenges and motivations for our work, and details some of the existing solutions to the problems at hand. In Section 3, we describe our data model and the notion of *significance*. The FussyTree data structure, its construction and querying are defined in Section 4. This is followed by Section 5, where we discuss evaluation strategies and metrics for our experiments. We report our experiments and findings in Section 6, and suggests extensions to our work in Section 7. We conclude with a discussion of the problem at hand in Section 8.

2. MOTIVATION AND CHALLENGES

2.1 Pervasiveness of Autocompletion

The concept of autocompletion has become prevalent in current user interfaces. It has found its way into mobile phones [38], OS-level file and web browsers [25], desktop search engines such as Google Desktop [22], GNOME Beagle [21] and Apple Spotlight [20], various integrated development environments, email clients such as Microsoft Outlook [23], and even word processors such as Microsoft Word [23] and OpenOffice [2]. The latest versions of Microsoft Internet Explorer [24] and Mozilla Firefox [25] implement autocompletion to suggest search queries to the user. This widespread adoption of autocompletion demands more from the backend mechanisms that support it in terms of fast response times with large amounts of suggestion data. While improvements in underlying computer hardware and software do allow for implementation of more responsive user interfaces, the basic computational challenges involving autocompletion still need to be solved.

2.2 Related Work

The concept of autocompletion is not new. Implementations such as the "Reactive Keyboard" developed by Darragh and Witten [6] have been available since the advent of modern user interfaces, where the interface attempts to predict future keystrokes based on past interactions. Learning-based assistive technologies have been very successfully used to help users with writing disabilities [18], and this technology is now being extended to accelerate text input for the average user.

There have been successful implementations for autocompletion in controlled language environments such as command line shells developed by Motoda and Yoshida [29], and by Davison and Hirsh [7]. Jacobs and Blockeel have addressed the problem of Unix command prediction using variable memory Markov models. Integrated Development environments such as Microsoft Visual Studio also exploit the limited nature of controlled languages to deliver accurate completion mechanisms that speed up user input.

As in these controlled languages, there is high predictability in natural human language text, as studied by Shannon [37], making a great case for the invention of automated natural language input. Phrase prediction and disambiguation for natural language have been active areas

of research in the speech recognition and signal processing community [11], providing reliable high-level domain knowledge to increase the accuracy of low-level audio processing. Phrase level prediction and disambiguation have also found great use in the area of machine translation to improve result quality [42]. Similar ideas involving language modelling have been used for spelling correction [17]. The exact task of sentence completion has been tackled using different approaches. Grabski et al. [12] have developed an information retrieval based set of techniques, using cosine similarity as a metric to retrieve sentences similar to the query. Bickel et al. [3] take on a more language model-based approach, by estimating parameters in linearly interpolated n-gram models.

While there have been significant advances in the level of phrase prediction for natural language, there appears to be very little attention paid to the *efficiency* of the prediction itself. Most models and systems are built for offline use, and are not designed for real time use at the speed of human typing - several queries a second. Also, the data models implemented by most are agnostic of actual implementation issues such as system memory. We attempt to bridge this gap, using a data-structures approach to solving this problem. We conceive the phrase completion problem as a suffix tree implementation, and institute methods and metrics to ensure result quality and fast query response times.

In addition to the nature of the queries, we recognize parameters that have not been considered before in traditional approaches to the problem setting. While the concept of contextual user interaction is quite prevalent [36], the *context* of the query phrase has been ignored in surveyed text-prediction literature. This is an important feature, since most human textual input is highly contextual in nature. In the following sections, we evaluate the ability of improving result quality using the query context.

The problem of frequent phrase detection has also been addressed in the context of "phrase browsing" in which documents are indexed according to a hierarchy of constituent phrases. The Sequitur algorithm used in these techniques [31, 28, 33] suffers from the problem of aggressive phrase creation and is not ideal for cases where we wish to index only the n-most frequent phrases, as opposed to all the repeated phrases in the document. Additionally the in-memory data structure created is hence extremely large and infeasible for large datasets.

Efficient data structures for prefix-text indexing have been studied. For example, trie variants such as burst tries [13, 43] have been shown to be effective for indexing word sequences in large corpora. However, these techniques still require memory resident structures and furthermore do not consider phrase boundaries or phrase frequencies, and hence cannot be used for our application.

2.3 Vocabulary and size

There has been considerable work [40, 41, 34] that considers the problem of frequent phrase storage using suffix trees for both small and large data sets. However, traditional techniques that study efficient suffix tree construction make an assumption that the vocabulary (tree alphabet) involved is small. Since we consider natural language phrase autocompletion as our target application, our vocabulary is that of all the possible words in the text, which is a very large number. Hence, the average fanout of each node in the suffix tree is expected to be quite high, that is, each word will have a high number of distinct words following it. In addition, the trees are expected to be sparse and extremely varied in their structure. This high fanout and sparse, varied structure challenges current scalable construction algorithms since sub-tasks in these algorithms, such as the identification of common substrings.

Such flat, wide suffix-trees also make querying difficult due to the increased costs in searching through the nodes at each level. A possible solution is to maintain each list of children in lexicographic order, which affects the lower bounds on suffix tree construction taking it from $O(n)$ to $O(n \log \sigma)$, where σ is the size of the vocabulary and n the size of our input text. Farach et al. [9] address this very problem and propose a linear-time algorithm for suffix tree construction. However, this algorithm involves multiple scans of the input string to construct and splice together parts of the tree, which makes it impossible to implement applications that are incremental or stream-like in nature, or where the corpus size is very large.

In other related work, the estimation of phrase frequencies has been looked into is an index estimation problem. Krishnan et al. [16] discuss the estimation techniques in the presence of wildcards, while Jagadish et al. [14] use various properties such as the short-memory property of text to estimate frequency for substrings. However, both papers consider only low vocabulary applications. For example, the construction of pruned count suffix trees, as suggested in these papers, is infeasible for phrases due to the large intermediate size of the trees, even with in-construction pruning.

3. DATA MODEL

In the context of the problem setting described above, we now formalize our autocompletion problem.

Let a document be represented as a sequence of words, $w_1, w_2, \ldots, w_N$. A phrase r in the document is an occurrence of consecutive words, $w_i, w_{i+1}, \ldots, w_{i+x-1}$, for any starting position i in $[1, N]$. We call x the *length* of phrase r, and write it as $len(r) = x$.

The autocompletion problem is, given $w_1, w_2, \ldots w_{i-1}$, to predict completions of the form: $w_i, w_{i+1}, \ldots, w_{i+c-1}$, where we would like the likelihood of this completion being correct to be as high as possible, and for the length of the completion to be as large as possible.

Of course, the entire document preceding word i can be very long, and most of it may have low relevance to the predicted completion. Therefore, we will consider a *phrase* to comprise a prefix of length p and a completion of length c. Values for both p and c will be determined experimentally.

We now introduce an example to help explain concepts throughout the rest of this paper, a sample of a multi document text stream for email as shown in Table 1. We also present the n-gram frequency table, in this example choosing a training sentence size $N = 2$. In other words, we determine frequencies for words and word pairs but not for word triples or longer sequences. We are not interested in phrases, or phrase components, that occur infrequently. In this example, we have set a threshold parameter $\tau = 2$. The italicized (last three) phrases have frequency below this threshold, and are hence ignored.

3.1 Significance

A phrase for us is any sequence of words, not necessarily a

Doc 1	please call me asap
Doc 2	please call if you
Doc 3	please call asap
Doc 4	if you call me asap

phrase	**freq**	**phrase**	**freq**
please	3	please call	3
call	4	call me	2
me	2	if you	2
if	2	me asap	2
you	2	*call if*	*1*
asap	3	*call asap*	*1*
		you call	*1*

Table 1: An example of a multi-document collection

grammatical construct. What this means is that there are no explicit phrase boundaries*. Determining such boundaries is a first requirement. In practice, what this means is that we have to decide how many words ahead we wish to predict in making a suggestion to the user.

On the one hand we could err in providing suggestions that are too specific; e.g. a certain prefix of the sentence is a valid completion and has a very high probability of being correct. However the entire suggestion in its completeness has a lower chance of being accepted. Conversely, the suggestions maybe too conservative, losing an opportunity to autocomplete a longer phrase.

We use the following definition to balance these requirements:

Definition 1. A phrase "AB" is said to be ***significant*** if it satisfies the following four conditions:

- *frequency* : The phrase AB occurs with a threshold frequency of at least τ in the corpus.
- *co-occurrence* : "AB" provides *additional information* over "A", i.e. its observed joint probability is higher than that of independent occurrence.

$$P(\text{“}AB\text{”}) > P(\text{“}A\text{”}) \cdot P(\text{“}B\text{”})$$

- *comparability* : "AB" has likelihood of occurrence that is "comparable" to "A". Let $z \geq 1$ be a comparability factor. We write this formally as:

$$P(\text{“}AB\text{”}) \geq \frac{1}{z} P(\text{“}A\text{”})$$

- *uniqueness* : For every choice of "C", "AB" is much more likely than "ABC". Let $y \geq 1$ be a *uniqueness* factor such that for all C,

$$P(\text{“}AB\text{”}) \geq yP(\text{“}ABC\text{”})$$

z and y are considered tuning parameters. Probabilities within a factor of z are considered comparable, and probabilities more than a factor of y apart are considered to be very different.

*Phrases are not expected to go across sentence boundaries, so the end of a sentence is an upper bound on the length of a phrase, but note that we have not yet seen the end of the current sentence at the time we perform phrase prediction, so we do not know what this upper bound is.

In our example, we set the frequency threshold $\tau = 2$, the comparability factor $z = 2$, and the uniqueness factor $y = 3$. Consider Doc. 1 in the document table. Notice that the phrase *"please call"* meets all three conditions of co-occurrence, comparability, and uniqueness and is a *significant* phrase. On the other hand, *"please call me"* fails to meet the uniqueness requirement, since *"please call me asap"* has the same frequency. In essence, we are trying to locate phrases which represent sequences of words that occur more frequently together than by chance, and cannot be extended to longer sequences without lowering the probability substantially. It is possible for multiple significant phrases to share the same prefix. E.g. both *"call me"* and *"call me asap"* are significant in the example above. This notion of significant phrases is central to providing effective suggestions for autocompletion.

4. THE FUSSYTREE

Suffix trees are widely used, and are ideal data structures to determine completions of given strings of characters. Since our concern is to find multi-word phrases, we propose to define a suffix tree data structure over an alphabet of words. Thus, each node in the tree represents a word. Such a *phrase completion suffix tree* is complementary with respect to a standard character-based suffix tree, which could still be used for intra-word completions using standard techniques. We refer to our data structure as a *FussyTree*, in that it is fussy about the strings added to it.

In this section we introduce the FussyTree as a variant of the pruned count suffix tree, particularly suited for phrase completion.

4.1 Data Structure

Since suffix trees can grow very large, a *pruned count suffix tree* [16] (PCST) is often suggested for applications such as ours. In such a tree, a count is maintained with each node, representing the number of times the phrase corresponding to the node occurs in the corpus. Only nodes with sufficiently high counts are retained, to obtain a significant savings in tree size by removing low count nodes that are not likely to matter for the results produced. We use a PCST data structure, where every node is the dictionary hash of the word and each path from the root to a leaf represents a frequent phrase. The depth of the tree is bounded by the size of the largest frequent phrase h, according to the h^{th} order Markov assumption that constrains w_t to be dependent on at most words $w_{t+1} \cdots w_{t+h}$. Since there are no given demarcations to signify the beginning and end of frequent phrases, we are required to store every possible frequent substring, which is clearly possible with suffix trees. Also, the structure allows for fast, constant-time retrieval of phrase completions, given fixed bounds on the maximum frequent phrase length and the number of suggestions per query.

Hence, a constructed tree has a single root, with all paths leading to a leaf being considered as frequent phrases. In the *Significance Fussytree* variant, we additionally add a marker to denote that the node is *significant*, to denote its importance in the tree. By setting the pruning count threshold to τ, we can assume that all nodes pruned are not significant in that they do not satisfy the frequency condition for significance. However, not all retained nodes are significant. Since we are only interested in significant phrases, we can prune

any leaf nodes of the ordinary PCST that are not significant. (We note that this is infrequent. By definition, a leaf node u has count greater than the pruning threshold and each of its children a count less than threshold. As such, it is likely that the uniqueness condition is satisfied. Also, since the count is high enough for any node not pruned, the comparability test is also likely to be satisfied. The co-occurrence test is satisfied by most pairs of consecutive words in natural language).

4.2 Querying: The *Probe* Function

The standard way to query a suffix tree is to begin at the root and traverse down, matching one query character in turn at each step. Let node u be the current node in this traversal at the point that the query is exhausted in that all of its characters have been matched. The sub-tree rooted at node u comprises all possible completions of the given query string.

For the FussyTree, we have to make a few modifications to the above procedure to take into account the lack of well-defined phrase boundaries. There are precisely two points of concern – we have to choose the beginning of the prefix phrase for our query as well as the end point of the completions.

Recall that no query is explicitly issued to our system – the user is typing away, and we have to decide how far back we would like to go to define the prefix of the current phrase, and use as the query to the suffix tree. This is a balancing act. If we choose too long a prefix, we will needlessly constrain our search by including irrelevant words from the "distant" past into the current phrase. If we choose too short a prefix, we may lose crucial information that should impact likely completions. Previous studies [14] have demonstrated a "short memory property" in natural language, suggesting that the probability distribution of a word is conditioned on the preceding 2-3 words, but not much more. In our scenario, we experimentally found 2 to be the optimum choice, as seen in section 6.5.2. So we always use the last two words typed in as the query for the suffix tree.

Once we have traversed to the node corresponding to the prefix, all descendants of this node are possible phrase completions. In fact, there are many additional phrase completions corresponding to nodes that have been pruned away in the threshold-based pruning. We wish to find not all possible completions, but only the significant ones. Since we have already marked nodes significant, this is a simple question of traversing the sub-tree and returning all significant nodes(which represent phrases) found. In general there will be more than one of them.

4.3 Tree Construction

4.3.1 Naive algorithm

The construction of suffix trees typically uses a *sliding window* approach. The corpus is considered a stream of word-level tokens in a sliding window of size N, where N is the order of the Markovian assumption. We consider each window instance as a separate string, and attempt to insert *all its prefixes* into the PCST. For every node that already exists in the tree, the node count is incremented by one. The tree is then *pruned* of infrequent items by deleting all nodes (and subsequent descendants) whose `count` is lower than the frequency threshold τ.

```
ADD-PHRASE(P)
1  while P ≠ {}
2  do
3     if IS-FREQUENT(P)
4       then
5            APPEND-TO-TREE(P)
6            return
7     REMOVE-RIGHTMOST-WORD(P)

IS-FREQUENT(P)
1  for i ← 1 to maxfreq
2  do
3     //slide through with a window of size i
4     for  each f in SLIDING-WINDOW(P, i)
5     do
6        if FREQUENT-TABLE-CONTAINS(f) = false
7          then
8               return false
9  return true

APPEND-TO-TREE(P)
1  //align phrase to relevant node in existing suffix tree
2  //add all the remaining words as a new path
```

4.3.2 Simple FussyTree Construction algorithm

The naive algorithm for PCST construction described above does not scale well since the entire suffix tree including infrequent phrases is constructed as an intermediate result, effectively storing a multiple of the corpus size in the tree. [16] suggests calling the `prune()` function at frequent intervals during the construction of the suffix tree using a scaled threshold parameter. However, our experiments showed that this strategy either produced highly inaccurate trees in the case of aggressive thresholding, or even the intermediate trees simply grew too large to be handled by the system with less aggressive thresholding.

To remedy this, we propose a basic *Simple FussyTree Construction* algorithm in which we separate the tree construction step into two parts. In the first step we use a sliding window to create an n-gram frequency table for the corpus, where $n = 1, ..., N$, such that N is the training sentence size. We then prune all phrases below the threshold parameter and store the rest. In our second step, we add phrases to the tree in a *fussy* manner, using another sliding window pass to scan through the corpus. This step stores all the prefixes of the window, given that the phrase exists in the frequency table generated in the first step. The testing of the phrase for frequency is done in an incremental manner, using the property that for any bigram to be frequent, both its constituent unigrams need to be frequent, and so on. Since the constituent n-grams are queried more than once for a given region due to the sliding window property of our insertion, we implement an LRU cache to optimize the `isFrequent()` function.

An example illustrating the FussyTree algorithm

We use the data from the example provided in Table 1 to construct our aggregate tokenized stream:

(please, call, me, asap, -:END:-, please, call, if, you, -:END:-, ···)

where the -:END:- marker implies a frequency of zero for any phrase containing it. We now begin the construction

of our tree using a sliding window of 4. The first phrase to be added is *(please, call, me, asap)* followed by its prefixes, *(please, call, me)*, *(please, call)* and *(please)*. We then shift the window to *(call, me, asap, -:END:-)*, and its prefixes *(call, me, asap)*, *(call, me)* and *(call)*, and so on. All phrases apart from those that contain either of *(call, if), (call, asap), (you, call)* or *(-:END:-)* will meet the frequency requirements, resulting in Fig. 1. Significant phrases are marked with a $*$. Note that all leaves are significant (due to the *uniqueness* clause in our definition for significance), but some internal nodes are significant too.

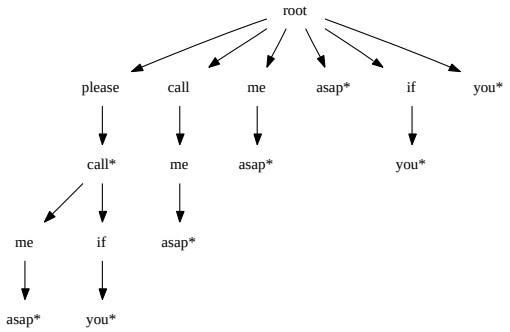


Figure 1: An example of a constructed suffix-tree

4.3.3 Analysis

We now provide a brief complexity analysis of our tree construction algorithm. It uses a sliding window approach over the corpus, adding the windowed phrases to the tree. Given a corpus of size S and a window of fixed size h, we perform frequency checks on each of the phrase incrementally. Hence, the worst case number of actual frequency checks performed is $S \times 2^h$. Note that h is a small number (usually under 8), and hence does not pose any cause for concern. Additionally, the LRU cache for frequency checks brings down this number greatly, guaranteeing that our corpus scanning is done in $O(S)$. Further, each phrase can in the worst case create a completely new branch for itself which can have a maximum depth of h, the size of the window. Thus, the number of tree operations is also linear with the corpus size, indicating that our construction algorithm time will grow linearly with the size of the corpus size. The constant-time lookup for the querying stage is explained thus: a query can at worst force us to walk down the longest branch in the tree, which is of maximum depth h, a fixed number.

We now look into the correctness of our tree construction with respect to our definition of significance. In our implementation, the FussyTree is constructed with only those phrases possessing a frequency greater than τ, ensuring that all nodes marked significant in the tree represent frequent phrases. This asserts correctness the *frequency* rule in the notion of significance. The *co-occurrence* and *comparability* are also rendered correctly, since we possess all the count information to mark them. However, to assert *uniqueness* in the leaf nodes of our tree, we note the loss of count information for the leaf nodes' children. Obviously, the frequency for the children of the leaf nodes is lesser than the threshold, and hence is most likely low enough to satisfy the uniqueness clause. However, there may be a case in which the children may have just missed the threshold, making the leaf node ineligible for the *uniqueness* clause. In this case, we conservatively mark all leaf nodes as significant.

4.3.4 Training sentence size determination

The choice of the size of the training sentence size N, is a significant parameter choice. The cost of tree construction goes up with N, as longer frequent phrases are considered during the construction. This predicates the need to store n-gram frequency tables (where $n = 1 \ldots N$) take up considerable resources, both in terms of storage and query times to check for frequency. The cost of phrase look-up in the resultant tree also increases because of this reason. Hence, in the interest of efficiency, we would like to minimize the value of N. Yet, using too small a value of N will induce errors in our frequency estimates, by ignoring crucial dependence information, and hence lead to poor choices of completions. Keeping this in mind, we experimentally derive the value of N in Sec. 6.5.1.

We note similar ideas in [41], where frequency information is recorded only if there is a significant divergence from the inferred conditional probabilities. However we point out that the frequency counts there are considered on a per-item basis, as opposed to our approach, which is much faster to execute since there are no requirements to do a recurring frequency check during the actual creation of the FussyTree structure.

4.4 Telescoping

Telescoping [26] is a very effective space compression method in suffix trees (and tries), and involves collapsing any single-child node (which has no siblings) into its parent node. In our case, since each node possesses a unique count, telescoping would result in a loss of information, and so cannot be performed directly, without loss of information. Given the large amount of storage required to keep all these counts, we seek techniques to reduce this, through mechanisms akin to telescoping.

To achieve this, we supplant the multi-byte `count` element by a single-bit flag to mark if a node is "significant" in each nodeat depth greater than one. All linear paths between significant nodes are considered safe to telescope. In other words, phrases that are not significant get ignored, when determining whether to telescope, and we do permit the collapsing together of parent and child nodes with somewhat different counts as long as the parent has no other significant children. To estimate the frequency of each phrase, we do not discard the count information from the significant nodes directly adjacent to the root node. By the *comparability* rule in the definition of significance, this provides an upper bound for the frequency of all nodes, which we use in place of the actual frequency for the `probe` function. We call the resulting telescoped structure a *Significance FussyTree with (offline) significance marking.*

Online significance marking

An obvious method to mark the significance of nodes would be to traverse the constructed suffix tree in a postprocessing step, testing & marking each node. However, this method requires an additional pass over the entire trie. To avoid them, we propose a heuristic to perform an on-the-fly marking of significance, which enables us to perform a significance based tree-compaction without having to traverse the entire tree again.

Given the addition of phrase $ABCDE$, only E is considered to be promoted of its flagged status using the current frequency estimates, and D is considered for demotion of its status. This ensures at least 2-word pipelining. In the case of the addition of $ABCXY$ to a path $ABCDE$ where E is significant, the branch point C is considered for flag promotion, and the immediate descendant significant nodes are considered for demotion. This ensures considering common parts of various phrases to be considered significant.

```
APPEND-TO-TREE(P)
1  //align phrase to relevant node in existing suffix tree
2     using cursor c, which will be last point of alignment
3  CONSIDER-PROMOTION(c)
4  //add all the remaining words as a new path
5     the cursor c is again the last point of alignment
6  CONSIDER-PROMOTION(c)
7  CONSIDER-DEMOTION(c → parent)
8  for each child in c → children
9    do
10       CONSIDER-DEMOTION(child)
```

We call the resulting structure a *Significance FussyTree with online significance marking.*

5. EVALUATION METRICS

Current phrase / sentence completion algorithms discussed in related work only consider single completions, hence the algorithms are evaluated using the following metrics:

$$Precision = \frac{\text{n(accepted completions)}}{\text{n(predicted completions)}}$$

$$Recall = \frac{\text{n(accepted completions)}}{\text{n(queries, i.e. initial word sequences)}}$$

where n(accepted completions) = number of accepted completions, and so on. In the light of multiple suggestions per query, the idea of an *accepted completion* is not boolean any more, and hence needs to be quantified. Since our results are a ranked list, we use a scoring metric based on the inverse rank of our results, similar to the idea of Mean and Total Reciprocal Rank scores described in [35], which are used widely in evaluation for information retrieval systems with ranked results. Also, if we envisage the interface as a drop-down list of suggestions, the *value* of each suggestion is inversely proportional to it's rank; since it requires 1 keypress to select the top rank option, 2 keypresses to select the second ranked one, and so on. Hence our precision and recall measures are redefined as:

$$Precision = \frac{\sum(1/\text{rank of accepted completion})}{\text{n(predicted completions)}}$$

$$Recall = \frac{\sum(1/\text{rank of accepted completion})}{\text{n(queries, i.e. initial word sequences)}}$$

To this end we propose an additional metric based on a "profit" model to quantify the number of keystrokes saved by the autocompletion suggestion. We define the income as a function of the length of the correct suggestion for a query, and the cost as a function of a *distraction cost d* and the rank of the suggestion. Hence, we define the **Total Profit Metric**(**TPM**) as:

$$TPM(d) = \frac{\sum(\text{sug. length} \times \text{isCorrect}) - (\text{d} + \text{rank})}{\text{length of document}}$$

Where *isCorrect* is a boolean value in our sliding window test, and d is the value of the distraction parameter. TPM metric measures the *effectiveness* of our suggestion mechanism while the precision and recall metrics refer to the quality of the suggestions themselves. Note that the distraction caused by the suggestion is expected to be completely unobtrusive to user input : the value given to the distraction parameter is subjective, and depends solely on how much a user is affected by having an autocompletion suggestion pop up while she is typing. The metric TPM(0) corresponds to a user who does not mind the distraction at all, and computes exactly the fraction of keystrokes saved as a result of the autocompletion. The ratio of TPM(0) to the total typed document length tells us what fraction of the human typing effort was eliminated as a result of autocompletion. TPM(1) is an extreme case where we consider every suggestion(right or wrong) to be a blocking factor that costs us one keystroke. We conjecture that the average, real-world user distraction value would be closer to 0 than 1.

We hence consider three main algorithms for comparison: (1) We take the naive pruned count suffix tree algorithm*(Basic PCST)* as our baseline algorithm. (2) We consider the basic version of the FussyTree construction algorithm called the *Simple FussyTree Construction* as an initial comparison, where we use frequency counts to construct our suffix tree. (3) Our third variant, *Significance FussyTree Construction* is used to create a significance-based FussyTree with significance marked using the online heuristic.

6. EXPERIMENTS

We use three different datasets for the evaluation with increasing degrees of size and language heterogeneity. The first is a subset of emails from the Enron corpus [15] related to emails sent by a single person which spans 366 emails or 250K characters. We use an email collection of multiple Enron employees, with 20,842 emails / 16M characters, as our second collection. We use a more heterogeneous random subset of the Wikipedia[†] for our third dataset comprising 40,000 documents / 53M characters. All documents were preprocessed and transformed from their native formats into lowercase plaintext ASCII. Special invalid tokens (invalid Unicode transformations, base64 fragments from email) were removed, as was all punctuation, so that we can concentrate on simple words for our analysis. All experiments were implemented in the Java programming language and run on a 3GHz x86 based computer with 2 gigabytes of memory and inexpensive disk I/O, running the Ubuntu Linux operating system.

To evaluate the phrase completion algorithms, we employ a sliding window based test-train strategy using a partitioned dataset. We consider multi-document corpora, aggregated and processed into a tokenized word stream. The sliding window approach works in the following manner: We assume a fixed window of size n, larger than or equal to the size of the training sentence size. We then scan through the corpus, using the first α words as the *query*, and the first β words preceding the window as our *context*. We then

[†]English Wikipedia: `http://en.wikipedia.org`

Algorithm	Small	Large Enron	Wikipedia
Basic PCST	13181	1592153	–
Simple FussyTree	16287	1024560	2073373
Offline Significance	19295	1143115	2342171
Online Significance	17210	1038358	2118618

Table 2: Construction Times (ms)

retrieve a ranked list of suggestions using the suggestion algorithm we are testing, and compare the predicted phrases against the remaining $n - \alpha$ words in the window, which is considered the "true" completion. A suggested completion is accepted if it is a prefix of the "true" completion.

6.1 Tree Construction

We consider the three algorithms described at the end of Sec. 5 for analysis. We ran the three algorithms over all three candidate corpora, testing for construction time, and tree size. We used the standard PCST construction algorithm [14] as a base line. We found the values of comparability parameter $z = 2$ and uniqueness parameter $y = 2$ in computing significance to be optimum in all cases we tested. Therefore, we use these values in all experiments reported. Additionally, the default threshold values for the three corpora were kept at 1.5×10^{-5} of corpus size; while the value for the training sentence size N was kept at 8, and the trees were queried with a prefix size of 2. The threshold, training sentence size and prefix size parameters were experimentally derived, as shown in the following sections.

We present the construction times for all three datasets in Table 2. The PCST data structure does not perform well for large sized data – the construction process failed to terminate after an overnight run for the Wikipedia dataset. The FussyTree algorithms scale much better with respect to data size, taking just over half an hour to construct the tree for the Wikipedia dataset. As we can see from Table 2, the Significance FussyTree algorithm is faster than offline significance marking on the Simple FussyTree, and is only slightly slower to construct than the Simple FussyTree algorithm.

6.2 Prediction Quality

We now evaluate the the prediction quality of our algorithms in the following manner. We simulate the typing of a user by using a sliding window of size 10 over the test corpus, using the first three words of the window as the context, the next two words as the prefix argument, and the remaining five as the *true completion*. We then call the probe function for each prefix, evaluating the suggestions against the true completion. If a suggestion is accepted, the sliding window is transposed accordingly to the last completed word, akin to standard typing behavior.

In Table 4 we present the TPM(0) and TPM(1) scores we obtained for three datasets along with the recall and precision of the suggestions. We compare our two tree construction algorithms, the Simple FussyTree, and the Significance FussyTree. The Basic PCST algorithm is not compared since it does not scale, and because the resultant tree would any way be similar to the tree in the Simple FussyTree algorithm. We see (from the TPM(0) score) that across all the techniques and all the data sets, we save approximately a fifth of key strokes typed. Given how much time each of us puts into to composing text every day, this is a really huge saving.

Algorithm	Small	Large Enron	Wikipedia
Simple FussyTree	8299	12498	25536
Sigf. FussyTree	4159	8035	14503

Table 3: Tree Sizes (in nodes) with default threshold values

Corpus: Enron Small

Dataset / Algo	Recall	Precision	TPM(0)	TPM(1)
Simple FussyTree	26.67%	80.17%	22.37%	18.09%
Sigf. FussyTree	43.24%	86.74%	21.66%	16.64%

Corpus: Enron Large

Dataset / Algo	Recall	Precision	TPM(0)	TPM(1)
Simple FussyTree	16.59%	83.10%	13.77%	8.03%
Sigf. FussyTree	26.58%	86.86%	11.75%	5.98%

Corpus: Wikipedia

Dataset / Algo	Recall	Precision	TPM(0)	TPM(1)
Simple FussyTree	28.71%	91.08%	17.26%	14.78%
Sigf. FussyTree	41.16%	93.19%	8.90%	4.6%

Table 4: Quality Metrics

To put the numbers in perspective, we consider modern word processors such as Microsoft Word [23], where autocompletion is done on a single-suggestion basis using a dictionary of named entities. We use the main text of the *Lord of The Rings* page on the English Wikipedia as our token corpus, spanning 43888 characters. In a very optimistic and unrealistic scenario, let us assume that *all* named entities on that page (for our problem, all text that is linked to another Wikipedia entry) are part of this dictionary; this could be automated using a named entity tagger, and also assume that we have a **perfect** prediction quality given a unit prefix size. Given this scenario, there are 226 multiple word named entity instances, which, given our optimistic assumptions, would result in **an ideal-case completion profit** of 2631 characters, giving us a $TPM(0)$ of **5.99**%. Our phrase completion techniques can do better than this by a factor of 3.

Despite the discarding of information through significance marking and telescoping, we observe how the *Significance FussyTree* algorithm results in a much smaller tree size (as seen in Table 3), and can still perform comparably with the baseline *Simple FussyTree* in terms of result quality. We observe an overall increase in recall and precision on adding significance, and that it causes a slight reduction in TPM metrics especially in high-variance corpora (as opposed to single author text). We consider the TPM drop to be acceptable in light of the reduction in tree size.

Looking across the data sets, we observe a few weak trends worth noticing. If we just look at the simple FussyTree scores for recall and precision, we find that there is a slight improvement as the size of the corpus increases. This is to be expected – the larger the corpus, the more robust our statistics. However, the TPM scores do not follow suit, and in fact show an inverse trend. We note that the difference between TPM and recall/precision is that the latter additionally takes into account the length of suggestions made (and accepted), whereas the former only considers their number and rank. What this suggests is that the average length of accepted suggestion is highest for the more focused corpus, with emails of a single individual, and that this length is lowest for the heterogeneous encyclopedia. Finally, the

Algorithm	Small	Large Enron	Wikipedia
Simple FussyTree	0.020	0.02	0.02
Sigf. FussyTree	0.021	0.22	0.20
Sigf. + POS	0.30	0.23	0.20

Table 5: Average Query Times (ms)

recall and precision trends are mixed for the significance algorithms. This is because these algorithms already take significance and length of suggestion into account, merging the two effects described above. As expected, the TPM trends are clear, and all in the same direction, for all the algorithms.

6.3 Response Time

We measure the average query response times for various algorithms / datasets. As per Table 5, it turns out that we are well within the 100ms time limit for "instantaneous" response, for all algorithms.

6.4 Online Significance Marking

We test the accuracy of the online significance marking heuristic by comparing the tree it generates against the offline-constructed gold standard. The quality metrics for all three datasets, as shown in Table 6, show that this method is near perfect in terms of accuracy[‡], yielding excellent precision[§] and recall[¶] scores in a shorter tree construction time.

Dataset	Precision	Recall	Accuracy
Enron Small	99.62%	97.86%	98.30%
Enron Large	99.57%	99.78%	99.39%
Wikipedia	100%	100%	100%

Table 6: Online Significance Marking heuristic quality, against offline baseline

6.5 Tuning Parameters

6.5.1 *Varying training sentence size*

As described in Sec. 4.3.4, the choice of the training sentence size N is crucial to the quality of the predictions. We vary the value of this training sentence size N while constructing the Significance FussyTree for the Small Enron dataset, and report the TPM scores in Fig. 2. We infer that the ideal value for N for this dataset is 8.

6.5.2 *Varying prefix length*

The prefix length in the probe function is a tuning parameter. We study the effects of suggestion quality across varying lengths of query prefixes, and present them in Fig. 2. We see that the quality of results is maximized at length = 2.

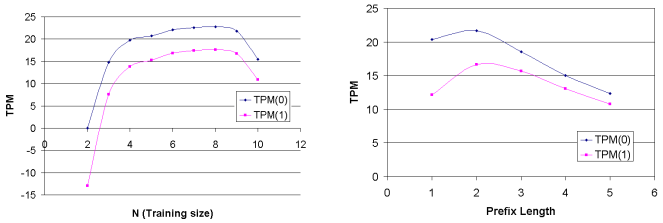
6.5.3 *Varying frequency thresholds*

To provide an insight into the size of the FussyTree, in nodes, as we vary the frequency threshold during the construction of the FussyTree for the Small Enron and Large Enron datasets; and observe the change in tree size, as shown in Fig.3.

[‡]percentage predictions that are correct

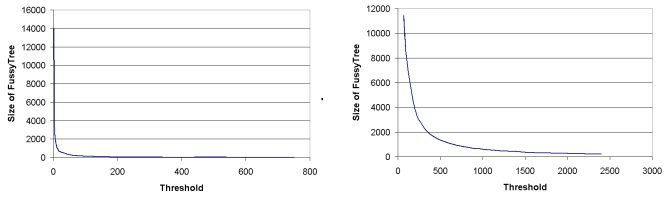
[§]percentage significant marked nodes that are correct

[¶]percentage of truly significant nodes correctly predicted



(a) (b)

Figure 2: Effect of varying (a) training sentence size (b) prefix length on TPM in Small Enron dataset



(a) (b)

Figure 3: Effect of varying threshold on FussyTree size in (a) Small Enron (b) Large Enron datasets

6.5.4 *Varying tree size*

We now use the frequency threshold as a way to scale up the size of the tree for the Significance FussyTree. We plot the TPM values(0 and 1) against the tree size for the Small Enron dataset. As can be seen, since the decrease in tree size initially causes a very gradual decrease in TPM, but soon begins dropping quite sharply at around threshold = 5 and above. Hence, we prefer to use threshold = 4 in this case.

7. POSSIBLE EXTENSIONS

In experiments over our test corpora in Sec. 6.3, we observe that the `probe` function takes less than a hundredth of a millisecond on average to execute and deliver results.

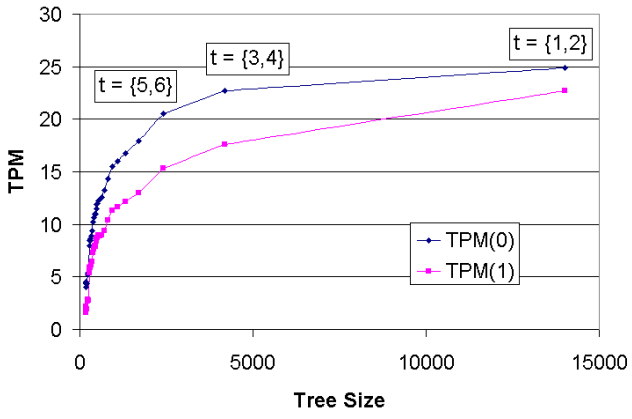


Figure 4: Effect of varying FussyTree size on TPM for Small Enron dataset, t labels show threshold values for the respective tree sizes

Corpus: Enron Small

Dataset / Algo	Recall	Precision	TPM(0)	TPM(1)
Simple + POS	25.0%	77.09%	22.22%	17.93%
Sigf. + POS	40.44%	81.13%	21.25%	16.23%

Corpus: Enron Large

Dataset / Algo	Recall	Precision	TPM(0)	TPM(1)
Simple + POS	16.13%	85.32%	15.05%	9.01%
Sigf. + POS	23.65%	84.99%	12.88%	6.16%

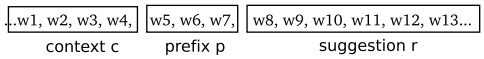
Corpus: Wikipedia

Dataset / Algo	Recall	Precision	TPM(0)	TPM(1)
Simple + POS	22.87%	87.66%	17.44%	14.96%
Sigf. + POS	41.16%	95.3%	8.88%	4.6%

Table 7: Quality Metrics, querying with reranking

Studies by Card [5] and Miller [27] have shown that the appearance of "instantaneous" results is achieved within a response time span of 100ms. Hence, this validates our premise and shows that there is a clear window of opportunity in terms of query time, to include mechanisms that improve result quality using the additional information. We discuss some methods to improve quality in this section.

We base our extensions on the premise that the *context* of a phrase affects the choice of suitable phrase completions. The *context* of a phrase is defined as the set of words $w_{i-y}, w_{i-y+1}, \ldots, w_{i-1}$ preceding the phrase $w_i, w_{i+1}, \ldots w_{i+x}$.



The *context* of a prefix can be used to determine additional semantic properties, such as semantic relatedness of the context to the suggestion, and the grammatical appropriateness based on a part-of-speech determination. We conjecture that these features abstract the language model in a way so as to efficiently rank suggestions by their validity in light of the context. This context of the query is used to *rerank* the suggestions provided by the FussyTree's `probe` function. The reranking takes place as a postprocessing step after the query evaluation.

7.1 Reranking using part-of-speech

We use a hash map based dictionary based on the Brill part-of-speech*(POS)* tagger [4] to map words to their part-of-speech. A part of speech automata is trained on the entire text during the sliding window tree construction, where each node is a part of speech, and each n-gram (to a certain length, say 5) is a distinct path, with the edge weights proportional to the frequency. Hence the phrase *"submit the annual reports"* would be considered as an increment to the *"verb-det-adj-noun"* path weights. The reasoning is that part-of-speech n-grams are assumed to be a good approximation for judging a good completion in running text. During the query, the *probe* function is modified to perform a successive `rerank` step, where the suggestions R are ranked in descending order of $prob(POS_c, POS_p, POS_{r_i})$, where POS_c, POS_p and POS_{r_i} are the POS of the context, prefix and suggestion, respectively. We then evaluate the precision, recall, TPM and execution time of the new reranking-based `probe` function, as reported in Table 7. We observe that the reranking has little average affect on Recall, Precision and TPM. We note however that on any individual query, the input can be significantly positive or negative. Isolating the cases where the effect is positive, we are working towards a characterization of the effect.

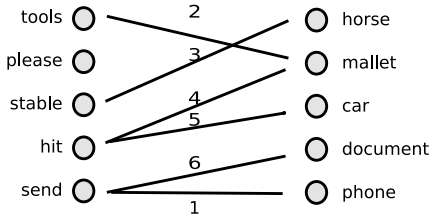


Figure 5: Bipartite graph - synset co-occurrences

7.2 Reranking using semantics

Another source of contextual information is the set of meanings of the prefix's word neighborhood. We reason that some word meanings would have a higher probability to co-occur with certain others. For example, for any sentence mentioning *"hammer"* in its context and *"the"* as prefix, it would more probable to have *"nail properly"* as a possible completion than *"documents attached"*, even if the latter is a much more frequent phrase overall, disregarding context. A simple bipartite classifier is used to consider $prob(A|B)$, where A is the set of WordNet [10] synsets, or word meanings in the result set, and B is the set of synsets in the query, as shown in Fig. 5. The classifier returns a collection of synsets for every synset set provided in the context and prefix. Suggestions are then ranked based on the number of synsets mapped from each suggestion. However, experiments show that the benefit due to semantic reranking was statistically insignificant, and no greater than due to POS reranking. In addition, it has resource requirements due to the large amount of memory required to store the synset classes*, and the high amount of computation required for each bipartite classification of synsets. We believe that this is because co-occurrence frequency computed with the prefix already contains most of the semantic information we seek. We thus do not consider semantic reranking as a possible improvement to our system.

7.3 One-pass algorithm

The FussyTree algorithm involves a frequency counting step before the actual construction of the data structure. This preprocessing step can be incorporated into the tree construction phase, using on-line frequency estimation techniques, such as those in [19, 8] to determine frequent phrases with good accuracy. A single pass algorithm also allows us to extend our application domain to stream-like text data, where all indexing occurs incrementally. We analyze various features of frequent phrases, such as part-of-speech, capitalization, and semantics. We show how the part-of-speech information of these phrases is a good feature for predicting frequent phrases with near 100% recall, by building a simple naive Bayesian classifier to predict frequency. Since the part-of-speech features of frequent strings are likely to be much less sensitive to changes in corpus than the strings themselves, we reason that it is viable to create a POS-based classifier using an initial bootstrap corpus. The classifier can

*WordNet 2.1 has 117597 synsets, and 207016 word-sense pairs

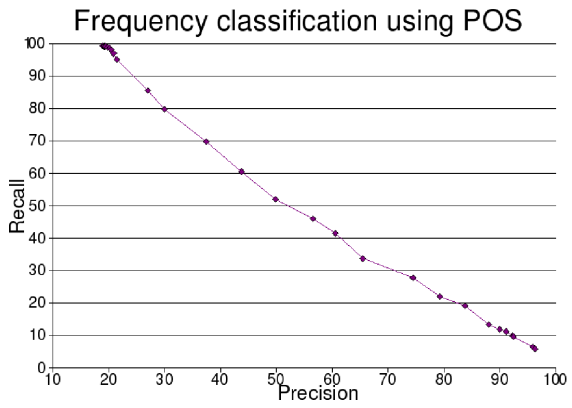


Figure 6: Recall vs precision for POS-based frequency classifier.

then be used to build suffix trees, while at the same time train itself. This is a very convenient technique to use for frequent re-indexing of data, for example an overnight re-index of all personal email that improves the autocompletion for the next day. Our experiment proceeds as follows. We train a Naive Bayes classifier to classify a phrase as *frequent* or *infrequent* based on the POS n-grams, which are generated using the process described in section 7.1. We then test the classifier on the training data, which in our case is the online trading emails from the Enron dataset. We also report numbers from tests on other datasets; first, a separate sample of emails from the Enron collection, and secondly the sample of text from the Wikipedia.

On the conservative side of the precision-recall curve, experiments on the large sized Enron collection report precision scores of 30%, 27% and 20%, for 80%, 85% and 99% recall respectively. We focus on this part of the recall-precision curve, since it is acceptable to have a large number of prospective frequent phrases, which are pruned later when updated with real counts.

8. CONCLUSION AND FUTURE WORK

We have introduced an effective technique for multi-word autocompletion. To do so, we have overcome multiple challenges. First, there is no fixed end-point for a multi-word "phrase". We have introduced the notion of significance to address this. Second, there often is more than one reasonable phrase completion at a point. We have established the need for ranking in completion results, and have introduced a framework for suggesting a ranked list of autocompletions, and developed efficient techniques to take the query context into account for this purpose. Third, there is no dictionary of possible multi-word phrases, and the number of possible combinations is extremely large. To this end, we have devised a novel FussyTree data structure, defined as a variant of a count suffix tree, along with a memory-efficient tree construction algorithm for this purpose. Finally, we have introduced a new evaluation metric, TPM, which measures the net benefit provided by an autocompletion system much better than the traditional measures of precision and recall. Our experiments show that the techniques presented in this paper can decrease the number of keystrokes typed by up to 20% for email composition and for developing an encyclopedia entry.

Today, there are widely used *word completion* algorithms, such as T9 [38]. We have shown that phrase completion can save at least as many keystrokes as word completion. However, word completion and phrase completion are complementary rather than competing. In terms of actual implementation, phrase completion can be triggered at every word boundary(e.g. when the user types a space), while word completion can be queried all other times.

As possible extensions, we discussed the notion of reranking suggestions using sentence context, and one-pass construction of the FussyTree. As future work, the idea of text autocompletion can be extended to various other applications. The current state of art in genomics only considers 1-8 bases as atomic units. There are no studies done at longer, variable strings of bases, which would require large-vocabulary capable suffix tree implementations. Another possible application is the inference of XML schema, which can be done by training the FussyTree with the XML paths as input. The single-pass version of the construction algorithm allows us to create a FussyTree for trend detection in streams, where the significance concept can be used to mark the start and end of various trends.

9. REFERENCES

[1] B. Bailey, J. Konstan, and J. Carlis. The Effects of Interruptions on Task Performance, Annoyance, and Anxiety in the User Interface. *Proceedings of INTERACT 2001*.

[2] C. Benson, M. Muller-Prove, and J. Mzourek. Professional usability in open source projects. *Conference on Human Factors in Computing Systems*, pages 1083–1084, 2004.

[3] S. Bickel, P. Haider, and T. Scheffer. Learning to Complete Sentences. *16th European Conference on Machine Learning (ECML05)*, pages 497–504, 2005.

[4] E. Brill. A simple rule-based part of speech tagger. *Proceedings of the Third Conference on Applied Natural Language Processing*, pages 152–155, 1992.

[5] 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 181–186, 1991.

[6] J. Darragh, I. Witten, and M. James. The Reactive Keyboard: a predictive typing aid. *Computer*, 23(11):41–49, 1990.

[7] B. Davison and H. Hirsh. Predicting sequences of user actions. *Notes of the AAAI/ICML 1998 Workshop on Predicting the Future: AI Approaches to Time-Series Analysis*, 1998.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 102–113, 2001.

[9] M. Farach. Optimal suffix tree construction with large alphabets. *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 137–143, 1997.

[10] C. Fellbaum. *Wordnet: An Electronic Lexical Database*. Bradford Book, 1998.

[11] E. Giachin and T. CSELT. Phrase bigrams for

continuous speech recognition. *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, 1, 1995.

[12] K. Grabski and T. Scheffer. Sentence completion. *Proceedings of the 27th annual international conference on Research and developement in information retrieval*, pages 433–439, 2004.

[13] S. Heinz, J. Zobel, and H. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002.

[14] H. Jagadish, R. Ng, and D. Srivastava. Substring selectivity estimation. *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 249–260, 1999.

[15] B. Klimt and Y. Yang. The Enron Corpus: A New Dataset for Email Classification Research. *Proceedings of the European Conference on Machine Learning*, 2004.

[16] 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 282–293, 1996.

[17] K. Kukich. Technique for automatically correcting words in text. *ACM Computing Surveys (CSUR)*, 24(4):377–439, 1992.

[18] C. MacArthur. Word Processing with Speech Synthesis and Word Prediction: Effects on the Dialogue Journal Writing of Students with Learning Disabilities. *Learning Disability Quarterly*, 21(2):151–166, 1998.

[19] G. Manku and R. Motwani. Approximate frequency counts over data streams. *Proceedings of the Twenty-Eighth International Conference on Very Large Data Bases*, 2002.

[20] Apple Inc. Spotlight Search `http://www.apple.com/macosx/features/spotlight`.

[21] Gnome Foundation. Beagle Desktop Search `http://www.gnome.org/projects/beagle/`.

[22] Google Inc. Google Desktop Search `http://desktop.google.com`.

[23] Microsoft Inc. Microsoft Office Suite `http://www.microsoft.com/office`.

[24] Microsoft Inc. Windows Explorer `http://www.microsoft.com/windows`.

[25] Mozilla Foundation. Mozilla Firefox version 2 `http://www.mozilla.com/firefox/`.

[26] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[27] R. Miller. Response time in man-computer conversational transactions. *Proceedings of the AFIPS Fall Joint Computer Conference*, 33:267–277, 1968.

[28] A. Moffat and R. Wan. Re-Store: A System for Compressing, Browsing, and Searching Large Documents. *Proceedings of the International Symposium on String Processing and Information Retrieval, IEEE Computer Society*, 2001.

[29] H. Motoda and K. Yoshida. Machine learning techniques to make computers easier to use. *Artificial Intelligence*, 103(1):295–321, 1998.

[30] B. Myers, S. Hudson, and R. Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28, 2000.

[31] C. Nevill-Manning, I. Witten, and G. Paynter. Browsing in digital libraries: a phrase-based approach. *Proceedings of the second ACM international conference on Digital libraries*, pages 230–236, 1997.

[32] L. Paulson. Building rich web applications with Ajax. *Computer*, 38(10):14–17, 2005.

[33] G. Paynter, I. Witten, S. Cunningham, and G. Buchanan. Scalable browsing for large collections: a case study. *Proceedings of the fifth ACM conference on Digital libraries*, pages 215–223, 2000.

[34] A. Pienimaki. Indexing Music Databases Using Automatic Extraction of Frequent Phrases. *Proceedings of the International Conference on Music Information Retrieval*, pages 25–30, 2002.

[35] D. Radev, H. Qi, H. Wu, and W. Fan. Evaluating Web-based Question Answering Systems. *Proceedings of LREC*, 2002.

[36] T. Selker and W. Burleson. Context-aware design and interaction in computer systems. *IBM Systems Journal*, 39(3):880–891, 2000.

[37] C. Shannon. Prediction and entropy of printed English. *Bell System Technical Journal*, 30(1):50–64, 1951.

[38] M. Silfverberg, I. S. MacKenzie, and P. Korhonen. Predicting text entry speed on mobile phones. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 9–16, New York, NY, USA, 2000. ACM Press.

[39] A. Taddei. Shell Choice A shell comparison. Technical report, Guide CN/DCI/162, CERN, Geneva, September 1994d.

[40] S. Tata, R. Hankins, and J. Patel. Practical Suffix Tree Construction. *Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August*, pages 36–47.

[41] S. Tata, J. Patel, J. Friedman, and A. Swaroop. Towards Declarative Querying for Biological Sequences. Technical report, Technical Report CSE-TR-508-05, University of Michigan, 2005.

[42] D. Vickrey, L. Biewald, M. Teyssier, and D. Koller. Word-Sense Disambiguation for Machine Translation. *HLT/EMNLP*, 2005.

[43] H. E. Williams, J. Zobel, and D. Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, 2004.