Predicting Source Code

Effectiveness of Prediction based Source Code Auto Completion.

Rahul Kapoor
r@trie.org

ABSTRACT
Auto Completion is the facility provided by most modern Integrated Development Environments and source code editors for word completion when editing source code. All auto completion mechanisms that we know of use syntactic knowledge of the programming language to provide this feature. We investigate the use of programming language agnostic prediction models to provide auto completion. We implement prediction based auto completion by using a variant of the popular compression scheme called Prediction by Partial Matching. We compare the results to an implementation of Syntax based Auto Completion. The results include a measurement of Key Strokes Per Character as well as subjective evaluation. We find that the prediction based auto completion performs remarkably well and consistently performs better than the syntax based completion method implemented.

Keywords
Auto Completion, Intellisense, Prediction, Integrated Development Environment, IDE, Editor

1. INTRODUCTION
Auto completion, also known by other names such as Content Assist, Code Completion and Intellisense is now ubiquitous in the world of source code editing. It is present in tightly integrated development environments (IDEs) which target a single language as well venerable general purpose text editors such as Vi and Emacs. All IDEs as well as general purpose editors that we know depend upon knowledge of the structure programming language to provide code completion. This syntax based approach uses knowledge of the underlying programming language. The knowledge of the underlying programming language includes the knowledge of the structure and the knowledge of available names (identifiers). The knowledge of the structure is usually obtained from the grammar of the language. The knowledge of the available names can be obtained using different mechanisms: Quite a few contemporary programming languages like Java, C# actually embed this information in the generated binary artifacts, while in the case of languages like C this information is usually obtained directly from the source. Implementations of Syntax based completion are therefore inherently tightly tied to a particular language.

The task of Syntax based auto completions is made difficult by the fact that the text being edited is not always in a syntactically valid state. The program being edited will eventually need to to be in a syntactically valid state to allow execution but intermediate forms are often not. Different Syntax based auto completions methods deal with different levels of grace with these situations. The code completions strategies even in modern IDE's will often time completely fail when the program being edited are in certain invalid states.

Apart from the actual program text, programs often contain significant portions of elements which are opaque from the point of view of the programming language. These include the content of literals in particular string literals and comments. Syntax based completion approaches usually do not suggest completions when inside these elements.

The Auto Completions approach introduced by this paper advocates the use of modern prediction algorithms for auto completion as opposed as well as to augment the traditional method described above. The prediction based approach is languages agnostic. Since the presented method uses machine learning, it can be used for any programming language as long as appropriate source code is available for training the model.

The prediction based approach also works well for all invalid intermediate states since it does not depend upon being able to parse the program text.

The prediction based approach can also suggest completions when editing the content of literals as well as comments.

The general notion of predictive interfaces has been well studied in the literature in two different contexts. The first is in the context mobile devices which have limited input capabilities. The second is systems based on augmentative and alternative communication theories, used to help people with physical or cognitive disabilities. For a survey of current state of text prediction see [6]. There are also multiple examples of using information theoretical methods for providing input to computers including the gesture based Dasher [13] and the metropolis keyboard [15]. [7] explores the application of predictive interfaces in a human computer interaction setting.

One measure used to evaluate text inputs methods is KSPC (introduced by MacKenzie in [10]) . KSPC is an an acronym for Keystrokes per characters. It is simply the number of keystrokes needed to generate a certain text. KSPC for a regular text entry using a Qwerty keyboard is naturally one (ignoring the complications of shift keys and meta keys). KSPC can be greater than one in the case of mobile devices which might need multiple key strokes to enter a single character. In the case of predictive text entry using a regular keyboard the KSPC should obviously be less than one. We use KSPC to compare the effectiveness of the different code completion implementations.

The completion mechanism implemented is based upon using the popular Prediction By Partial Matching [4] , variant C algorithms (PPMC). PPMC is one of the best know compression algorithms for general purpose use. We also implement a syntax based completion provider to compare the results of the prediction based completion scheme.

The paper is organized in the following manner. In Section 2 we detail the implemented code completion methods. Specifically we detail the working of PPMC algorithm in a prediction setting and the rules upon which the syntax based completion scheme is implemented. We also describe how both the prediction based method can be used to augment the syntax based scheme. Section 3 introduces the model used to measure the effectiveness of the completion schemes impemented. In 4 we present the results of our experiments using the implemented code completion methods. Sections 5 and 6 present directions for further work and the conclusion respectively.

2. COMPLETION METHODS

In this section we present the implementation details of the completion schemes implemented. All training and completion analysis is done on the scrubbed version of the source code. Scrubbing compresses all white space, removes all comments and removes all string literals. We first describe PPMC based completion method followed by the syntactic approach and then a approach which combines the two in a natural manner. All experiments in this paper are performed using source code from the Java programming language [9].

2.1 PPMC based completion

Prediction By Partial Match is a well known loss less compression algorithm. All loss less compression can be abstracted as consisting of two stages sometimes termed as the model and the coder. In [2] Blelloch defines the model as the "component that somehow captures the the probability distribution of messages by knowing or discovering something about the structure of the input". The coder part uses the model to encode the input in a manner that saves space. The problem addressed by the model component in lossless data compression is same as the problem that needs to be solved for prediction. Therefore every loss less compression algorithm can be in theory used for predictions [1]

PPM belongs to a class of compression algorithms generally termed as a variable order Markov Model algorithm. A Markov model defines a system with a set of states, which moves from state over a period of time. [1] As the system moves from state to state, the probability of moving to a certain state is determined only by the resent state transitions. The order of a markov model is the number of previous states that determine the next state: for example in a first order markov model the next state solely depends on the current state. An n^{th} order markov model is one which the next state depends only upon the previous n states.

Like most prediction algorithms the actual process can be logically broken down into two phases, the learning phase which is followed by the prediction phase. In the learning phase PPM builds a markov model based upon the sequence of characters seen. More formally if the sequence of characters is from an alphabet Σ, the model is trained using sequence of characters from Σ, t of length j after which the model can be used to answer queries of the form $P(\sigma|s), \sigma \in \Sigma, s \in \{\Sigma\}^n$ that is you can query the model for probability of any character from the alphabet Σ given that you have just seen the sequence s of length n. This string s is often termed as the context. In the PPM scheme the maximum order M of the markov model constructed has to be specified before the learning phase. [2] The method is termed as a variable order because if a n length context does not lead to a prediction then a $n-1$ length context is used and so on till the zeroth order model is reached.

The working of the PPMC method is best explained using an example. Consider the training sequence *wabbawabbawoo* over the alphabet $\Sigma = \{a, b, o, w\}$. After training the model can be used to answer queries of the form $P(\sigma|s)$ where $\sigma, s \in \Sigma, |s| \leq M$. Under PPMC $P(\sigma|s)$ is given by:

$$P(\sigma|s) = \frac{N(s\sigma)}{|\Sigma_s| + \sum_{\sigma' \in \Sigma_s} N(s\sigma')} \tag{1}$$

In equation 1 $N(s\sigma)$ denotes the number of times $s\sigma$ occurs in the training sequence. Σ_s denotes the set of symbols appearing after s in the training sequence. Using equation 1 $P(b|ab) = \frac{N(abb)}{|\Sigma_{ab}| + \sum_{\sigma' \in \Sigma_{ab}} N(s\sigma')}$ for the example training sequence $N(abb) = 2, \Sigma_{ab} = \{b\}$ which gives $P(b|ab) = \frac{2}{1+2} = \frac{2}{3}$.

The required probabilities for PPMC can also be conveniently calculated by building a trie. Each node in the trie has a symbol and a counter associated with it. The maximum depth of the tree is $M + 1$. The trie starts with a root node representing the empty context e. Whenever a new symbol is read its M length context is added to the trie if it does not exist. If some part of the path does exist in the trie

[1]It is convenient to think of this process varying with time and in reality the variance could be any step such as the next input in a stream of characters.

[2]A variant of PPM called PPM* allows unbounded contexts.

already, the counters along the path are incremented. Table 1 shows the first few steps in building this trie for our example string *wabbawabbawoo*. Figure 1 shows the complete trie after the complete training sequence has been read. The counters along the trie nodes can be used to find $N(s\sigma)$ as well as Σ_s. Using the counters in the trie nodes from figure 1 the probability $P(b|ab)$ is easily calculated to be $\frac{2}{1+2} = \frac{2}{3}$. The tree can be built up in time linear in the length of the training sequence and takes space $O(M.n)$ where n is the length of the training sequence in the worst case.

The formula given for calculating the probability by equation 1 gives the probability 0 whenever $N(s\sigma) = 0$ that is whenever $s\sigma$ does not appear in the training sequence its probability is zero. This problem is known as the zero frequency problem. This problem is handled in PPMC via the escape mechanism. The escape mechanism works by allocating probability mass $P(escape|s)$ for all symbols in the alphabet which do not appear after the context s and the remaining $1 - P(escape|s)$ among symbols that do appear after the context. The recursive relation ship for probability which leads to the variable order portion of the markov model is given for context $s, |s| < M$ by equation 2:

$$P(\sigma|s) = \begin{cases} P'(\sigma|s) & s\sigma \in t \\ P'(escape|s) \times P(\sigma|s_{-1}) & s\sigma \notin t \end{cases} \tag{2}$$

The notation s_{-1} in equation 2 refers to the sub string obtained from s by removing the first symbol, $s\sigma \in t$ implies that s was presented in the training sequence t and $s\sigma \notin t$ implies that $s\sigma$ was not present in the training sequence. Including the escape mechanism the probabilities are given by equations 3 and 4. This particular implementation of the escape mechanism is due to Moffat [11].

$$P'(\sigma|s) = \frac{N(s\sigma)}{|\Sigma_s| + \sum_{\sigma' \in \Sigma_s} N(s\sigma')} \tag{3}$$

$$P'(escape|s) = \frac{|\Sigma_s|}{|\Sigma_s| + \sum_{\sigma' \in \Sigma_s} N(s\sigma')} \tag{4}$$

Equations 2, 3 and 4 can be used along with the counters present in the trie nodes to easily calculate the required probabilities.

PPMC also makes use of another optimization usually termed the exclusion mechanism. The exclusion mechanism allows for more accurate estimates when the escape mechanism is used. The exclusion mechanism is described in detail in [1] and [11].

Another implementation detail is the probability assignment to consider when the recursive relation ship given by equation 2 leads to the zeroth order model. One simple mechanism is assign equal probability $\frac{1}{\Sigma}$ to each symbol. Another, better approach is to use the frequency distribution of the symbols seen as the zeroth order model. This leads to the problem that you may get zero probability in case some symbols are absent from the training sequence. This can be easily dealt with by using laplace smoothing when calculating the frequencies distribution.

Another implementation detail is the handling of counter over flow. Counter over flows are dealt with by re-scaling all counter values in case a counter overflows. Re-scaling is done by dividing all counters by a constant to lower all values. No counter values are allowed to go below one during the re scaling process.

Once the PPMC based completion implementation is trained using the training data, it is ready to be used for predictions. The built up markov model is used to predict up to a constant number of characters on the basis of the current context. The model can predict multiple words at the same time as well as partial words. Symbols are predicted till the probability falls below a specified threshold or a maximum length is reached. The model could be used to make more than one prediction but as implemented the PPMC based completion only makes a single prediction. As results later show this does not seem to be major shortcoming.

2.2 Syntax Based Completion

To be able to compare the effectiveness of prediction based completion method to Syntax aware completion, we implemented a Syntax based completion scheme for the java programming language. This completion scheme uses the knowledge of the structure of the Java programming language as well as knowledge of the names available from all needed libraries referenced by the source, as well as all compilation units needed to compile the file being edited. The completion also uses conventions not enforced by the language but which are almost univesally followed in the Java world. The syntax based completions, as is typical in most editors and IDE's is limited to single words in each prediction. Also as is typical usually all predictions matching the context are presented to the user.

The major rules implemented are briefly described below:

2.2.1 Package Rule

Java files usually start a package declaration. The package declaration is followed by a package name. A Package declaration can only appear at the start of a compilation unit. There can only be at most one package declaration.

2.2.2 Import Rule

The key word import is followed by a type name or part of a package name. The import keyword is followed by a package qualified type name or a partial package name and a wild card.

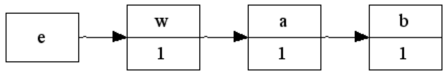
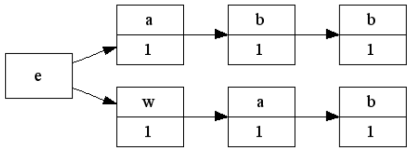
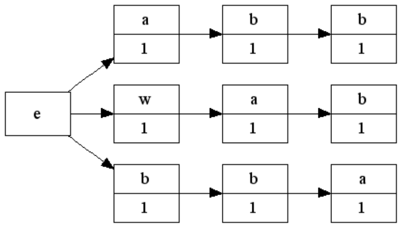
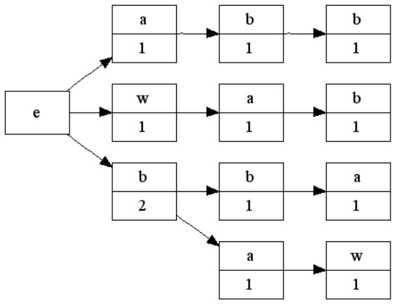
2.2.3 Extends Rule

The keyword extends is followed by a class name. The class name may be a fully specified name with a package name or a shorter version (see the imports rule later).

2.2.4 Implemented Rule

The keyword implements is followed by an interface name. The interface name may be fully specified with a package name or a shorter version (see the imports rule later).

Table 1: Steps in building trie for *wabbawabbwoo*

Sequence read so far	trie
wab	
wabb	
wabba	
wabbaw	

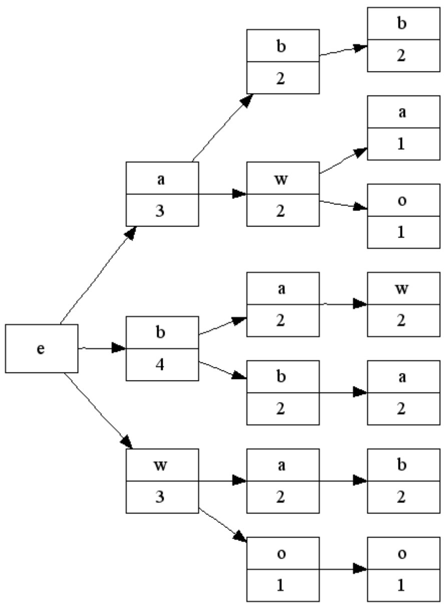


Figure 1: The complete trie for *wabbawabbwoo*

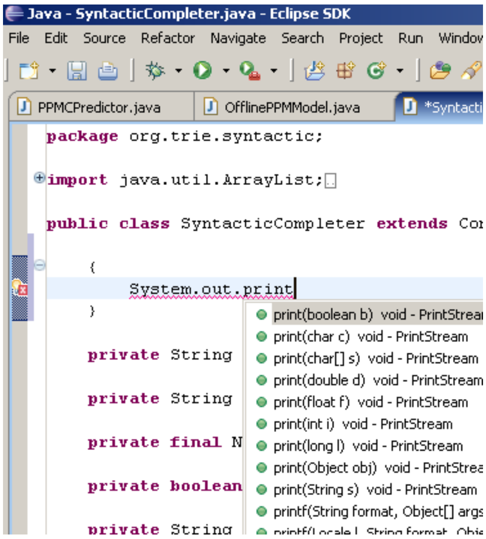


Figure 2: Completions in the Eclipse IDE for `System.out.p`

2.2.5 *Imports Rule*

When searching for applicable types, the imports declaration, the members in the same package as the current type as well as the implicit import of the java.lang package is taken into account.

2.2.6 *Member Access*

The . operator in Java is the member access operator and the completions suggested can be limited to the members of the type of the operand of the . operator. The members for a type are listed using the Java reflection library. This rule is also used for static member access for a type.

2.2.7 *Type Name Heuristic*

Word starting with upper case characters usually refer to type names.

2.2.8 *Default Rule*

When none of the above rules are applicable the default suggestions include all language keywords and primitive types.

2.3 Combined Approach

In the Syntax based approach mentioned in above, all predictions made are presented to the user in lexicographic order. This is how usually how most Syntax based predictions work. An example of this approach is shown figure 2. The figure shows the completion available to the user after he has typed `System.out.p`. This lexicographical presentation is not too useful and can be vastly improved by presenting the most probable choices towards the top of the list. The mixed completions strategy does precisely that buy combining the information from the PPMC model with the suggested completions. Since the PPMC completion method may also predict more than one word at a time in which case the suggested completion is also added to the top of the suggested completions list.

3. MEASUREMENT MODEL

This section describes the idealized model used for measurement of effectiveness of the different implemented code completions methods. The basic data used for comparison is the KSPC metric. The model reflects a simple setting, assuming a perfect user. The perfect user starts writing the source code starting from the beginning of the file and continuing on till the end. The user never edits any text and makes no mistakes. The assumed user also always selects a prediction if it saves him keystrokes. It is assumed that no special key strokes are needed to present the list of possible completions, that is the ordered list of completions is available after each character typed by the user. The number of key strokes needed to select an item suggested in the i^{th} place in the list of completions is i. This assumption corresponds with the real world action of using some sort of key stroke to move down the list in the presented order.

The model does not allow for the use of a mouse or any other pointing device. Nowhere does the model account for the cognitive load on the user having to mentally scan the list to check if the required completion is present. The model also does not take into consideration any notion of time. This highly simplified view of the world has the benefit of being amicable to objective KSPC measurement. We denote the total number of keystrokes which actually resulted in a character being typed by N_c. N_p is the total number keystrokes used to select a presented prediction. Note that a prediction is selected only if results in net key stroke savings for the user. The total number of characters in the final document is given by N_t. KSPC in this model is simply calculated using equation 5.

$$KSPC = \frac{N_c + N_p}{N_t} \tag{5}$$

4. RESULTS

We conducted experiments on multiple open source java projects. Results on all selected open source Java projects were very similar. Here we only reports results for the GNU Trove open source collections library [5]. For the Syntax based completions strategy a data base of available names has to be generated. This database is generated using the ctags [8] program. Whenever a list of accessible members for a type is needed the Java Runtime type information discovery facility (reflection) is used. The members for a type are lazily generated on demand. The ctags based named database is generated from all files in the project as well as for all referenced libraries. All the source files were scrubbed using the procedure mentioned in section 2. The prediction based completions use the scrubbed source files for training. A single test file is selected uniformly at random from all the scrubbed source files. This test file is excluded from training as well as ctag generation. N_c, N_p and N_t mentioned in section 3 are calculated using a driver program. The numbers are calculated for the pure prediction strategy, the syntax based completion strategy and the mixed strategy. The result across all projects were very similar. The average weighted by the number of characters in the file is calculated for each method. The weighted average for the three methods if given in table 2. The data used to calculate the average is presented in tables 3 and 4. The weighted average KSPC of the prediction method 0.485 is remarkably higher

Table 2: Weighted average for KSPC

Method	KSPC
PPMC	0.485
Syntax Based	0.691
Mixed	0.476

```
package gnu.trove;

final class ToObjectArrayProcedure<T> implements TObjectProcedure<T> {
    private final T[] target;
    private int pos = 0;

    public ToObjectArrayProcedure(final T[] target) {
        this.target = target;
    }
```

Figure 3: Predictions Based Completions for a portion of a test file

then the Syntax based method which has a KSPC of 0.691. Combining the two does not yield any significant improvement over the pure prediction based completions scheme. The KSPC achieved for the mixed scheme is 0.476.

To intuit the stellar performance of the prediction based scheme, it is insightful to examine the completions provided for a portion of a Java source file. Figure 3 shows the completions accepted by the ideal user in the model when using PPMC based completions. Figure 4 shows the completions accepted by the model user when presented with Syntax driven completions. In both the figures the bold portions reflect the portions the user did not have to type while the rest represents the text typed by the user. Figure 4 for the Syntax based completion does not reveal how many extra key strokes were needed to select the appropriate completions, making it look better in the figure than it actually was.

Some of the reasons the prediction based strategy performs much better than the completions strategy can be understood from the subjective evaluation of the completions. One of the major factor is that the prediction strategy often presents fairly long completions consisting of multiple words. PPMC based completion also suggests parts of words, while syntax based methods always suggests complete words only. Another reason is that the prediction strategy can actually suggest names for identifiers when defining them while the Syntax based strategy depends upon names being already defined. Another huge factor is the way predictions are implemented they only suggest one possible completion making the cost of accepting a prediction really small.

The fact that the prediction based completion only suggests one completion for any context is actually a major benefit

```
package gnu.trove;

final class ToObjectArrayProcedure<T> implements TObjectProcedure<T> {
    private final T[] target;
    private int pos = 0;

    public ToObjectArrayProcedure(final T[] target) {
        this.target = target;
    }
```

Figure 4: Syntax Based Completions for a portion of a test file

of the implementation. A single good suggestion is worth a lot more than many not as appropriate suggestion usually arising from Syntax based completions. With a real human user involved addition of each new prediction to the completion list also increases search time for the user by about 150 ms [6].

The Syntax based completion implemented could possibly more structure. The amount of information used during Syntax based editing is typical of the type used in general purpose editors, but some highly language specific editors like IntelliJ in case of Java actually use more heuristics and more knowledge of the structure of the programming language.

5. FURTHER WORK

In this paper we implemented a PPMC based code completion mechanism. The point of view of analysis of effectiveness was only key stroke savings. No attention was paid to efficiency in terms of time. Before a real implementation can be created the time efficiency of prediction will need to investigated. We also used only of one many possible completion algorithms. Other good options from the same family of variable order markov models include Probabilistic Suffix Trees [12] and CTW [14]. PPM also has a variant which can deal with context of unbounded length called PPM* [3]. One of the main benefits of prediction scheme is the language agnostic nature but we only implemented results for the Java programming language. The quality of prediction possible depends on the structure of the programming language and the style of the source code used for training. The authors are currently working on comparison of predictability of code across languages. The Syntax based completion implemented lags behind the state of art completion mechanism present in highly language specific editors (IntelliJ and Eclipse in case of the Java programming language). It would be an interesting experiment to compare the prediction based approach to completion provided by these editors.

From a key stroke saving perspective prediction seems to works well, but the authors realize that code completion provides more than that. Functions provided by code completion include quick access to documentation as well as code discovery. Developers often use code completion as a light weight code browser to figure out the facilities provides by a module or a class.

6. CONCLUSIONS

The PPMC based prediction model works remarkably well for providing code completions, easily outperforming the syntax based completion model implemented. This paper shows that from a keystroke saving perspective a prediction based model is worth looking into for the developers of IDE's and editors even if they already implement Syntax based code completion. The benefits of prediction include partial word completion, suggestion of names before being defined, possible completions for string literals and inside comments and ease with which they cope with invalid intermediate states produced during editing. A prediction only completion component would also make a great addition to a general purpose editor allowing out of the box autocompletion for all languages after a brief training step.

Table 3: KSPC Details for test files

File	N_t	PPMC $N_c + N_p$	Syntax Based $N_c + N_p$	Mixed $N_c + N_p$
TLinkedList.java	6286	3344	4445	3332
ToObjectArrayProcedure.java	291	129	212	121
THashIterator.java	684	229	430	185
TObjectObjectProcedure.java	105	35	72	28
HashFunctions.java	516	250	316	247
Timer.java	152	52	93	48
TObjectHashIterator.java	287	107	212	108
TObjectIdentityHashingStrategy.java	261	92	180	91
TObjectFunction.java	91	28	63	29
Tlinkable.java	248	64	140	59

Table 4: KSPC for various test files

File	PPMC	Syntax Based	Mixed
TLinkedList.java	0.532	0.707	0.530
ToObjectArrayProcedure.java	0.443	0.729	0.416
THashIterator.java	0.335	0.629	0.270
TObjectObjectProcedure.java	0.333	0.686	0.267
HashFunctions.java	0.484	0.612	0.479
Timer.java	0.342	0.612	0.316
TObjectHashIterator.java	0.373	0.739	0.376
TObjectIdentityHashingStrategy.java	0.352	0.690	0.349
TObjectFunction.java	0.308	0.692	0.319
Tlinkable.java	0.258	0.565	0.238

7. REFERENCES

[1] R. Begleiter, R. El-Yaniv, and G. Yona. On prediction using variable order markov models. Technical Report CS-2004-07, Technion - Israel Institute of Technology, May 2004.

[2] G. Blelloch. Introduction to data compression. October 2001.

[3] J. G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *The Computer Journal*, 40(2/3):67–??, ???? 1997.

[4] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32(4):396–402, April 1984.

[5] E. D. Friedman. Gnu trove: High performance collections for java.

[6] N. Garay-Vitoria and J. Abascal. Text prediction systems: a survey. *Univers. Access Inf. Soc.*, 4(3):188–203, 2006.

[7] S. Greenberg, J. J. Darragh, D. Maulsby, and I. H. Witten. Predictive interfaces: what will they think of next? pages 103–140, 1995.

[8] D. Hiebert. Exuberant ctags.

[9] B. Joy, G. Steele, J. Gosling, and G. Bracha. *Java(TM) Language Specification (2nd Edition)*. Addison-Wesley Pub Co, June 2000.

[10] I. S. MacKenzie. Kspc (keystrokes per character) as a characteristic of text entry techniques. In *Mobile HCI '02: Proceedings of the 4th International Symposium on Mobile Human-Computer Interaction*, pages 195–210, London, UK, 2002. Springer-Verlag.

[11] A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38:1917–1921, 1990.

[12] D. Ron, Y. Singer, and N. Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25(2-3):117–149, 1996.

[13] D. J. Ward, A. F. Blackwell, and D. J. C. MacKay. Dasher - a data entry interface using continuous gestures and language models. In *UIST*, pages 129–137, 2000.

[14] F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens. The context-tree weighting method: basic properties. *IEEE Trans. Info. Theory*, pages 653–664, 1995.

[15] S. Zhai, M. Hunter, and B. A. Smith. The metropolis keyboard - an exploration of quantitative techniques for virtual keyboard design. In *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 119–128, New York, NY, USA, 2000. ACM Press.