Japanese word prediction

Fredrik Lindh
frlindh@kth.se
850320-0290
073-6479325

Japanese studies, Lund University, Sweden.
Degree project in Japanese and linguistics, First Level.

Supervisors
Lars Larm (Lars.Larm@ostas.lu.se)
Arthur Holmer (Arthur.Holmer@ling.su.se)

Referat

Denna uppsats beskriver en implementation av en japansk ordprediktor skriven av författaren. Eftersom en ordprediktor inte verkar existera för Japanska, så kan den bli värdefull som ett mjukvaruverktyg inom assistiv teknik och kommunikation (AAC). Den största fördelen som ett sådant system för med sig är förbättrad skrivhastighet, samt att färre tangenttryckningar krävs för att producera text. Ordprediktion ställs ofta i kontrast mot ordkomplettering; den teknologi som T9-systemet i många mobiltelefoner och intellisense-motorer är baserade på. Det finns dock en skillnad då ordprediktion handlar om att föreslå ett följande ord då ett ord skrivs klart, mot att avsluta ett ord som håller på att skrivas. De fungerar oftast genom att tillhandahålla en lista över bigramsfrekvenser viktade med användarens preferenser och korpusdata. Ordprediktion tillsammans med ordkomplettering är bland de kraftfullaste assistiva verktyg som finns för att hjälpa funktionsnedsatta med deras dagliga kommunikationsbehov.

Huvudmålen för denna uppsats är:
1. Att röna ut de skillnader som uppstår i implementationen jämfört med andra språk
2. Undersöka vad som kvarstår att göra, både inom prototypen i sig och i allmänhet.
3. Skapa en fungerande **prototyp** av programmet för Japanska.

All kod i projektet är fritt tillgänglig och ligger för tillfället på:
http://www.mediafire.com/?rrhqtqsgp6ei6m3

Abstract

This report deals with the implementation of a Japanese word prediction engine written by the author. As this type of software does not seem to exist for Japanese at the time of writing, it could prove useful in Japanese augmentative and alternative communication (AAC) as a software tool used to improve typing speed and reduce the amount of keystrokes needed to produce text. Word prediction, in contrast to the word completion software commonly found in mobile phones and word processor intellisense engines etc. is a technique for suggesting a followup word after a word has just been completed. This is usually done by providing a list of the most probable words to the user, sorted by commonality (general and user-specific frequency). Combined with good word completion software and a responsive user interface, word prediction is one of the most powerful assistive tools available to movement impaired users today.

The main goals of the thesis will be to:
1. Answer as many of the questions raised by the language differences as possible.
2. Investigate further avenues of research in the subject.
3. Make a functional word prediction **prototype** for Japanese.

All project code is in the public domain and is currently hosted at:
http://www.mediafire.com/?rrhqtqsgp6ei6m3

I would also like to thank Akino Kuwahara for checking my Japanese,
and Ragnar Mjelde for being helpful in regards to the Tobii SonoScribe software.

Contents

1. Introduction

1.1 Term glossary

This thesis is aimed at linguists and computer linguists in particular, but recognizing that the terminology used might not be familiar to all readers, I have opted to include this glossary over technical terms used in the following chapters.

Heuristic
A heuristic is a method to optimize the way one might do something. In the context of this thesis, a heuristic can be thought of as some way to make word prediction go either faster or become more accurate by applying some known characteristic of Japanese vocabulary, grammar or other linguistic knowledge as a feature to the program.

Henkan
Is translating characters from one writing system to another on a computer. This can for example be performed on a windows machine with the Japanese IME installed by changing the language to Japanese / Kana input and then typing some text. Then press space and select some transformed input in the appearing combo box. By doing this, you have performed *henkan* on the input text.

Data structure
A data structure is simply some way of storing data on a computer. Popular ones include data lists, stacks, queues and tables.

Hash table
Is a type of data structure also known as a dictionary structure. This is because it works much like a dictionary where you have a *key* to a *value* object. The key is used to look up the value object in the data dictionary, analogous to using the index in a common dictionary. The main components of the software use this data structure for data lookup.

Serialization / marshaling
This is the technical term for saving and retrieving data to and from the hard drive.

Token
A token is an object consisting of a word and its meta data. This data can be many kinds of things, such as its part of speech, what topic it belongs to, what particle it accepts, how many times it has appeared in some predetermined scope and so on.

n-gram
are a general linguistic concept of treating *n* words as the same token. For

example, a bi-gram would be the token consisting of two words and their eventual meta data, a tri-gram would be the three-word counterpart.

POS tagging
Part of speech tagging is used for a multitude of reasons in natural language processing. Knowing what part of speech a particular word belongs to can help understand its role in the sentence it is in. Or help the program understand its semantic role, or even the words around it (using support vector machine techniques). It is also valuable in many n-gram analysis situations.

Parsing
Is the act of exploring and processing text. The processing could be almost anything but mainly pertains to syntactic analysis. It might involve preparing the text for some other process (preprocessing), or creating tokens from the text and loading them into some data structure. Or even mining the text for semantic relations.

1.2 Background

One of the more defining traits of the modern man compared to the other fauna of the earth is our ability to use highly complex language. This is not without problems however, since using audio signals for communication is both error prone and difficult to store. Written language eventually allowed us to do all these things and more, especially for speech and hearing impaired individuals; the access to a writing system actually *gave them a chance to participate in the general social forum*. Now, while it is true that literacy didn't become widespread before the 19th century[1], it has always provided an additional dimension of freedom to those who practice it. With the advent of modern society, literacy is approaching 85% world wide, and we in the Western world have a new companion to which we communicate almost exclusively with written language in some form or the other. I speak of course of computers. As of late, computers don't only provide us with an extended interface with which to communicate with each other over vast distances as though we were next to each other, they also give us the ability to put previously theoretical knowledge in areas such as mathematics, linguistics and physics into *practical use* with their immense computing power. Especially linguistics has seen a real jump in application areas in the last few years such AACS (augmentative and alternative communication systems), word completion, expert systems, information retrieval systems and many more. This thesis will deal with a certain type of AACS system called a word prediction engine, which can be of great use to functionally impaired users, especially when coupled with other cutting edge technology such as eye tracking and word completion.

1.3 Problem statement

There are quite a few problems to solve when developing word prediction software. This thesis will explore these problems using a primitive implementation written by the author as basis for the study, and suggest possible solutions to the problems encountered. The main components involved in the project are:

- The corpus used as the bigram data source
- The architecture of the software in question
- The computer linguistic problems local to the Japanese language
- Optimizations and heuristics

The thesis will concentrate on the three first components and only briefly touch the fourth since it's not really basal part of the problem and more of a means to improve the implementation.

1.4 A brief explanation of word prediction

Word prediction, not to be confused with word completion is a relatively new application area. Although both of the technologies are used to speed up typing speed the implementations and effects are quite different. Word completion deals with prediction which word the user wants to type *now*. It starts operating as soon as the user has typed the first letter in a word, analyzing the letters and sometimes earlier context to determine which word the user is trying to type. Good examples of word completion engines are the common T9 systems found in commercial mobile phones, the word completion found in word processors such as MS Office, and engines in integrated development environments such as the intellisense functionality in MS visual studio.
Word / phrase prediction on the other hand deals with the *next* word/words the user will want to write. It is especially useful for movement impaired users who need help writing very common social phrases[2] fast and often. Instances of this type of software is harder to find, but the Tobii SonoScribe Communicator suite is one good example.

Even though these two technologies do well on their own, they really are the most effective when used together, reaching upwards 60-80% of keystrokes saved[3], depending on the user and usage scenario. Word completion alone reduces the amount of keystrokes needed by 50-60%[4]. While this can seem like a lot even to the casual user, the perceived impact on the AAC prime users is many times more since this is their *only mode of communication*. Imagine being able to speak more than twice as fast when talking to your friends or co-workers every day!

1.5 Other word prediction software

There exist several commercial (e.g. the software used in Tobii Communicator suite) as well as non-commercial (e.g. the FASTY[5] project) word predictors today. There are several reasons why the author chose to not adapt one of the existing ones to Japanese compared to writing a custom implementation, the most relevant ones being:

- Many word prediction systems are completely integrated into their parent systems, especially when it comes to proprietary systems, for example the Tobii Communicator suite.
- Proprietary systems are ruled out completely because of monetary reasons; the project does not possess the funds to acquire such a license.
- All the systems found were more or less domain specific. The only really promising technology was the FASTY word prediction engine which is an EU-founded project aimed at creating a modular word prediction engine with multiple language support, but the underlying engine still works with the same parsing rules common to all Western languages, and it seems that the only way to resolve some of that major problems that comes of that (for example whitespace problems, more on that later in the thesis) is to write a new grammatical module, which is far too complex for one person to do under the time contraints. Using FASTY on the other hand would enable the use of many powerful heuristics, so exploring the possibility of writing a custom parsing filter for the technology might be a good future project.
- Implementing a new feature into an existing technology is usually not a simple undertaking. The same is true for NLP systems, and making a stand alone implementation for educational purposes is more often than not faster than the alternative since it avoids a lot of overhead work (including reading documentation, familiarizing oneself with the new system, finding all the quirks, etc. etc.). Simply put, the author did not feel comfortable pursuing this alternative given the time constraints and localization problems.

2. Tools

2.1 The .net framework

The entire implementation is written using the .net framework developed by Microsoft based on the common language infrastructure (CLI) in Windows. The framework was chosen due to its automatic memory management capabilities which enables more agile software development and less test time. The framework also has an excellent library of stock data structures and GUI creation tool sets which saves a lot of time developing an interface to the main prediction engine. There are other reasons for choosing the framework as well, such as one-click deployment and a first class IDE to name a few. The only real drawback is that it *only runs on Microsoft-compliant hardware and operating systems*. But since Microsoft still holds ~90% market share[6] and the thesis only aims at completing and documenting a prototype, it's still acceptable.

2.2 MeCab

MeCab is a Japanese Parts-Of-Speech-Tagger developed by Yuuichi Teranishi and supports both n-best POS tagging[7], *wakachigaki*, encoding conversion and several output modes as well as being platform independent. The only function I'll be using in my implementation however is the *wakachigaki* functionality since I suspect that some corpus elements might not be correctly formatted. Its use could be extended to using POS meta data in the heuristics of the program. That lies outside of the ambition scope of this thesis however and will remain an anecdote for now.
The tagger software uses conditional random fields[8] which is a statistical method of inferring relationships between words. It then uses this information to determine which word class is the most likely one for each word in the data set. The tagger is based on the older Japanese POS-tagger ChaSen. Both implementations utilize the IPA corpus, while MeCab also makes use of the Juman corpus as well as the Canna dic project.

2.3 The Tanaka corpus

The Tanaka corpus is a corpus consisting of ~150.000 Japanese/English sentence pairs compiled by professor Yasuhito Tanaka at Hyogo University and his students in and before 2001, and was then later refined by a number of professionals and volunteers. The corpus is currently hosted by the Tatoeba project and is used as a source of example sentences for the WWWJDIC translation service. The corpus was chosen due to it being freely available and having a very consistent form which makes it easy to programmatically parse. It's also relatively big which also made it a better candidate for the thesis. If licensing time had not been an issue, a more comprehensive corpus such as the Tokutei corpus might have been sought after instead. The Tanaka corpus is not without its problems however. The quality of the sentences is oftentimes very shoddy since they come from student assignments. Some of them are taken from old English-textbooks, song lyrics or machine translated literature. Since this makes the sentence structure differ a bit from natural spoken Japanese, it will of course also affect the prediction accuracy of the prototype since the prototype database will be built upon the statistical relationships present in the corpus. The documentation even warns against using the corpus for statistical analysis due to the risk of obtaining skewed data. But since the goal of the project is not to build a commercial grade predictor, the good qualities of the corpus still outweighs the bad.

2.4 Tobii Communicator / SonoScribe

The word prediction engine will be roughly modeled after the one present in Tobii Technology's Tobii Communicator Suite On Screen Keyboard Interpreter called "SonoScribe"[9]. The Tobii Communicator Suite is the main product of Tobii's assistive technology branch and is aimed at helping movement impaired users with their day-to-day communication needs by providing speech synthesis and other tools. The feature interesting to this thesis is the extensive support for word / phrase completion and prediction coupled with a touch and eye control interface. The graphical user interface lets the user type letters into the main display box by using any of its input methods and then uses its prediction engine to supply the user with the statistically most probable followup words and phrases compiled from corpora.
The implementation will utilize some of the features present in the SonoScribe software, mainly n-best suggestion methods and the statistical model used, but will omit the GUI and localization as well as almost all of the heuristics.

3. Problems in Japanese word prediction

3.1 The *wakachigaki* problem

One of the most difficult natural language processing (NLP) problems in Japanese is the so called *wakachigaki*[10] (分かち書き) problem. In most Western languages, words are delimited by white spaces and punctuation, which helps a lot with artificial parsing of text. Computers know when to stop and read a token from its current data set whenever it encounters a white space or period. Having this crutch has helped early NLP development immensely over the years by speeding up the general parsing process. Our Western white spaces are not completely without problems however. Ambiguity over so called "compound words" is aggravated because of increased use of white space over using hyphens which continues to cause computers some duress. For example, it is ambiguous whether the sentence:

I like chocolate chip cookies

is trying to convey a taste for slim chocolate cookies or a taste for the modern chocolate-chip filled cookies. Taking current trends into consideration, a human would most likely vote for the second alternative, but a computer has no way to determine this by just looking at the sentence itself. Some languages, like Swedish for example, is harsher when it comes to compounding words, writing them in series without using delimiters at all. A translation of the English sentence above would in consequence look like something in the lines of:

Jag gillar chokladflarnskakor

Notice how the last three words in English are compounded into one when written in Swedish. This helps a computer to correctly treat "Chocolate chip cookie" as its own token, but it also poses a new delimitation problem since compounding with this type of agglutinative characteristic gives rise to a combinatorial explosion of possible word permutations. Giving a logical token to each such compound word would be impractical for statistical reasons (More tokens means less increments per token which means less reliable results, not to mention the extreme memory requirements!) and nonsensical since the semantics and classification of a word like skomakargesällsarbetarorganisation (cobbler apprentices' workers union) is unclear and it would be better in this case to delimit it similar to the English translation to gain more usable bigram[11] data.

The delimitation problems in Japanese relevant to this thesis is similar to the ones presented in Swedish compound words, but the problem does not simply extend to long word compounds, but to the entire Japanese writing system. Japanese, like most other Eastern languages do not delimit their text with white space and Japan did not even use punctuation before the Meiji era (around the start of the 20th century)[12]. This means that before any type of machine driven parsing can take place, delimitation of tokens first have to been performed. That is where *wakachigaki* algorithms come into the picture. Like most NLP problems, there are several ways to approach the problem. One way is to split the text into manageable chunks and then crawl those chunks one character at a time to determine which words are *possible* with the help of a lexicon. A Viterbi search[13] can then be performed to determine the most likely distribution and sequence of words. Since Japanese now uses punctuation, this makes for an excellent basis of the initial chunking. Another welcome feature pertaining to the *wakachigaki* problem in Japanese that stands in contrast to other Asian languages is the combination of *kanji* (Japanese ideographic characters) and *kana* (The syllabic Japanese alphabet) used in common text. This is because there are certain patterns that can be exploited by a *wakachigaki* engine. One of these patterns is the use of so called "*okurigana*"; *kana* placed after a series of *kanji* that act as conjugation. Another very common pattern is to identify singular *kana* in front of a *kanji* compound as a particle. The presence of *okurigana* and particles leads to a syntax that regularly switches between *kana* and *kanji*. A *kanji* character after a *kana* character consequently usually means that a new token has begun. *Wakachigaki* will be used in this thesis to tokenize some parts of the corpus in preparation for further parsing.

3.2 Verb stemming

Another problem in parsing is verb stemming. Japanese has a very rich agglutinative conjugation system utilizing *kana* after verbs to add grammatical markers such as tense and modality. The following example from Tsujimura[(1999) p257] illustrates the conciseness of the Japanese particle and inflection system very well:

Taroo-ga	*Hanako-ni*	*Ziroo-o*	*Mitiko-ni*	*aw-ase-sase-ru.*
Taro-NOM	*Hanako-DAT*	*Jiro-ACC*	*Michiko-DAT*	*meet-cause-CAUS-PRES*

"Taro will cause (make/let) Hanako to cause Jiro to meet Michiko."

Even though most stemming operations deal with inflections, there are also verb to verb stemming involved when dealing with Japanese, as in the following example from Tsujimura[(1999) p297] where the verbs "eat" and "begin" are compounded.

Tabe-hajime-ru
eat- begin- PRES
To begin eating

Stemming these verbs is necessary to correctly give correct word suggestions. The most common method is to use a multi-pass system[14] to reduce complete verbs in steps. The hardest part is not the stemming however, but reproducing the correct verb form when suggesting words, since conjugation rules are highly context bound. As the prototype written for this thesis does not use any context analysis heuristics whatsoever, the GUI tackles this problem in a way similar to the primary suggestion engine. It will simply provide an n-best list of results, presenting the statistically most probable choice at the top. Also, the current incarnation does not use the multi-pass system described here but instead stems its verbs using corpus meta-data and simply cutting *kanji* compounds where the *okurigana* starts. One major improvement would be to change this to the multipass system, but to do this, changes in the *okurigana* suggestion pipeline would also have to be made, as well as the underlying data structures. This is because the current implementation only saves *complete* sequences of *okurigana*, not the individual morphemes. A concrete example would be that the verb tabesaseraremashita (食べさせられました) is saved as:

tabe	saseraremashita
eat-	*suffix*

and not as the more desirable

tabe	sase	rare	mashi	ta
eat-	*CAUS*	*PASS*	*HON*	*PAST*

3.3 Particle identification and placement

Identifying particles and associating them with their parent words is the next major problem to be dealt with. Japanese particles come in a variety of different types, some comparable to Western counterparts, like the standard locative, temporal and conjunctive particles with the main difference being that Japanese particles are post-positions in contrast to for example the pre-positions in English. Some of the more interesting particles in Japanese are their modal and case varieties. I've listed two examples of those from Tsujimura[(2007) p122] below. The case particles are of special interest for semantic parsing since they mark sentence constituents.

Modal example:

Goji-made	darou
five-to	probably

“It is probably until 5 o'clock”

Case particle example:

Ziroo-ga	Yosio-ni	ringo-o	age-ta
Ziro-NOM	Yoshio-DAT	apple-ACC	give-PAST

“Jiro gave an apple to Yoshio”

Since Japanese employs post-positions, a reverse parsing pattern appeared the most natural. This means that sentences is parsed back-to-forth. When a particle is encountered, it is saved until its parent word is encountered. Parsing in reverse also provides the additional benefit of making the verb stemming easier, allowing the program to prune the *okurigana* and then couple the stem with its particle and conjugations.

Another problem with particles local to the chosen corpus is that because of the colloquial nature of the sentences, variations in the transcribed pronunciations makes identifying particles hard in some cases. For example, the interrogative particle ka (か) is sometimes transcribed as kaa (かぁ) or even kaaa (かぁぁ) due to spoken emphasis. The only way to prune these words correctly would be to either make a stop list associating every variation with its base form, or use POS meta data. That is, MeCab or another similar software could be used to tag the particles, and even though most of them would be treated as unknown words, chances are pretty good that they would be correctly tagged as particles due to their syntactic position and bigram relationships. If they were correctly tagged, pattern matching could then take place to decide which particle it is.

3.4 *Kanji*, *kana* and IME *henkan*

Finally, some major differences in Western and Eastern input methods pose additional problems. Japanese uses 4 different character sets:

- *Romaji* (ローマ字), which is our roman alphabet. Used for foreign names and words.
- *Hiragana* (平仮名) is a syllabic alphabet used for native words and conjugation.
- *Katakana* (片仮名) has the same set of phonemes as *hiragana*, but is used for technical terms and loan words.
- *Kanji* (漢字) is a ideographic alphabet used interchangeably with *hiragana* when the writer wants to disambiguate a word or add formality. The use of *kanji* tightly coupled with Japanese culture. For example, an adult who doesn't use *kanji* is looked down upon as childish.

Keyboards are more often than not modeled after the English alphabet. This is also true in Japan, where most computers use the English model coupled with a so called Input Mode Editor (IME), even though keyboards accommodating the Japanese syllabic alphabet exist. Input is created by first writing the desired Japanese phonetically in roman letters, and then converting that text into Japanese characters by pressing a certain button. This process of input conversion is called *henkan* (変換) in Japanese and creates a few additional problems. The first problem is that since Japanese uses several parallel alphabets, the prototype would have to save all tokens either in one form, or in some type of type-agnostic format. The best approach would be the type-agnostic approach, maybe by representing the words in roman characters, and saving references to possible transcriptions, or do the conversions with the help of an IME application programming interface (API).

Trying to do automatic programmatical conversions using an API leads us to our next problem; not all Japanese character conversion operations are bijective! While conversions between *romaji*, *hiragana* and katanana all are, *kana* to *kanji* conversion is not. This is because the *kana->kanji* relationship

isn't injective. In other words, there is no unambiguous way to convert *kana* into *kanji* since most *kana*-sequences can be converted into *several different kanji* compounds. This does not mean that the reverse is true though, *kanji->kana* conversion is for most intents and purposes injective. So what does this mean for our prototype? While it is true that the lack of injective transforms hinders for example *kana* input conversion, there are a few ways to try to disambiguate *kana* into *kanji*. The first and foremost is using an input method editor. Using such a software solves the conversion problem, but it still doesn't solve the actual disambiguation problem (IME's can provide a list of possible conversions, but it can not decide which suggestion is the most probable one depending on context, and they are generally not good at *wakachigaki*). One possible solution might be to check each suggestion from the IME, coupled with its preceding word, against the relationship database to see which is the most probable combination.
Using this technique might provide simple support for *kana* input conversion, but using *kanji* when writing will probably still be more effective.

One other interesting feature of *kanji* is their idiomatic nature. *Kanji* are used in two ways in Japanese. The first one is as stems for native words which are then followed up by *okurigana*. The other use is as sino-Japanese compounds where they usually indicate a more formal version of a native word. One example can be demonstrated using the following three words:
とる (to ru)、採る (to ru)、採取する (sai shu su ru).
The first two words are pronounced exactly the same. The only difference is that the second one uses a *kanji* to replace the first "to". This can be done for several different reasons, one being to *disambiguate* the *kana*-sequence. If the *kanji* is not used, the two *kana* could mean a lot of different things, for example 盗る(to steal)、取る(to take something)、撮る(to take a picture)、録る (to record a video), among others. But if the *kanji* is used, some ambiguity is removed and the possible meanings are narrowed down to for example: "to pick [e.g. a flower]" or "to catch [an insect]". Some ambiguity still remains though, as it can still also mean "to take [an attitude] in addition to a few other, very context related uses. The third variation of the word however is very specific and no matter the context just means "to pick [a flower]" or "to collect [an insect]". If this unambiguity of *kanji* could be harnessed in for example word prediction software, a lot of the decision trees involved could be pruned, leading to better heuristic performance. But it could also be used for other useful things such as semantic parsing.

3.5 An overview of linguistic heuristics

There are many different ways to improve prediction speed and accuracy. These methods are collectively known as heuristics and mainly come in the form of learning and data processing algorithms. Most of the algorithms used in natural language processing belong to one of the following basic categories. Most of them are detailed in Jurafsky & Martin.

Statistical heuristics (used in this project) [J&M P.910, 178, 208]
Encompasses methods that build upon statistical relationships between words in the training set. A word is promoted if the words related to it exhibit favorable characteristics and downgraded if they do not. Some things that can be analyzed are for example: n-gram frequency, word frequency, phrase frequency, morphological frequency (the ratio of certain morphemes) and so on. Many implementations use n-gram Markov models.
Example: If word A appears more often than word B, word A should be prioritized.

POS tagging with n-gram analysis [J&M P. 108, 157, 167,178, 181, 208, 218]
Is a type of statistical heuristic, but the training set is first part-of-speech tagged before analyzed. The training set is therefore classified in a more general and grammatical way instead of the standard lexical way. This is useful when researching text characteristics and in language comparisons. Many implementations use n-gram Markov models and the Viterbi algorithm.
Example: If a word A has several different possible tags, the n-grams associated with that word/tag can be used to decide which of the possible tags is the most probable.

Learning heuristics [J&M P.122, 265]
There are many kinds of learning heuristics. The most popular ones range from simple user frequency lists (the system keeps a record of the user's favorite words and promote them appropriately) to advanced cloud based internet services hooked up to automated internet spiders that synchronizes a NLP system with current language trends as seen online.
Example: If a word A in the system has higher precedence than word B but the user has typed word B more than A to some predefined degree, word B might be suggested anyways.

Topic guidance [J&M P.147, 502, 824]
Most discussions have topic. This heuristic works by annotating every word in its dictionary with a topic tag. It then analyzes preexisting text and tries to decide which topic is overrepresented in the text. It then promotes words belonging to that topic to the user.
Example: If the system splits all words into texts into topic domains and finds that one domain is overrepresented, words from that domain might be suggested with higher precedence compared to other domains.

Word stemming (used in this project) [J&M P. 80, 102, 806]
Every word would become their own token in a system without a stemming heuristic. This is bad for several reasons, more tokens means a larger search space, and therefore lower accuracy. It also makes sense to sort semantic tokens by stem when suggesting words (it becomes almost nonsensical to not do so).
Example: Two words that share the same stem (i.e. Eat, eaten) can be reduced to a single token (i.e. eat-) with the help of stemming, instead of retaining the two words as two separate tokens (eat, eaten).

Stop list heuristics (used in this project) [J&M P.806]
We sometimes want to avoid parsing certain words in a training set, these might be meta-data or some other word we're not interested in. It therefore makes sense to create a stop list heuristic to skip those words.
Example: Suppose a corpora contains a meta-data marker to indicate that a sentence has low quality (The sentence header contains the marker {[LQ]}). We could either just remove the marker if we add it to a word stop list (suppose we still want the low quality sentence in the training set, but we don't want the marker itself). Or we could implement a sentence stop list that removes the entire sentence when it finds a meta-data marker in the list.

Voting heuristics (usually using OTS software) [Beáta M.]
A system that is effective and demanded a lot of its hardware ten years ago might only require a fraction of the computational power of a modern machine. This fact can be exploited by employing the voting heuristic, where the same problem is solved by several separate algorithms, choosing the best one (or an aggregate of all solutions) as the answer candidate.
Example: Suppose we have three different POS taggers. Further assume that two out of three of them tag a certain word as tag A, and the third tags the word as tag B. The rule of majority suggests that tag A should be chosen.

Support vector machines [J&M P.237]
Is a method used to find similarities in words and can be used as a heuristic to give hints of the nature of a certain token. In the context of NLP, in particular in conjunction with the POS statistical heuristic, to decrease search space, the method can be used to answer questions such as “what [kind of] words appear near to this word?”. This could be used to resolve an ambiguous token, by looking at it's support vector machine meta data.
Example: Suppose that a word A is to be semantically parsed and we have to resolve A between two possible candidates. We can determine which of these are more likely by examining the support vector for the current word to the support vectors of the two candidates and choosing the closest one (usually using dot product, the closer the dot product of a pair is to zero, the closer the vectors are).

Particle heuristics [J&M P.160, 435]
Particles can, to some degree, help disambiguate the meaning of a word, so keeping track of which word classes a certain particle accepts can easily eliminate a number of different tokens. Particles in Japanese are very useful especially in semantic parsing as they clearly point out sentence constituents such as the sentence topic (ha|は), the subject (ga|が) and the accusative object (wo|を).
Example: Suppose a sentence in the training set is:
"Thomas built a floor by himself."
The article a only accepts nouns, so the meaning of floor must be the noun (in contrast to the verb "to floor".

4. Program Implementation

4.1 Program architecture

Most of the design choices were made to accommodate the time-constraints present for the project. Since the goal was to both make a working prototype and to produce this thesis, the foremost concern was to complete a fully functional program within a couple of weeks and to concentrate on this text after that while making adjustments to the software. This of course impacted heavily on both the chosen base design as well as the heuristics used.

The implementation has a few core components:

- The preprocessing pipeline
- The corpus loading pipeline
- The back end relational database
- The input processing and word suggestion pipeline
- The UI

These components would be necessary for any implementation of a word predictor, and I've tried to design the general architecture and the different components to be as modular as possible to allow for easy replacement of any one component. The components themselves are all hooked up to some part of the UI and are therefore triggered by one of the several buttons there. So if anyone would like to replace an algorithm, using the related events handlers would be a good entry point. Be aware that even though the components themselves are not very tightly coupled, the *theme* of the components are. This means that, for example, changing from a bigram to a trigram database is not as nicely decoupled[15] as letting a simple change in the parsing algorithm do the trick, since the database would not support it. This should however be familiar to any programmer willing to give the code a go. The following pages will go over each of the components in more detail.

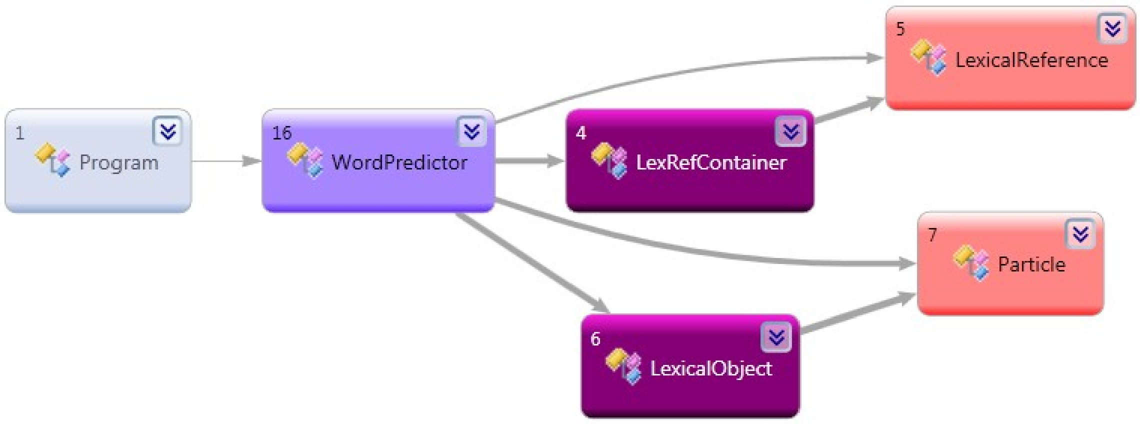
4.2 Corpus preprocessing and loading

There are many different flavors of corpora available both free online and as proprietary products, for a variety of languages. This means that finding a corpus for a project like this isn't very hard; the problem is that the style and quality of corpora vary with each instance. In other words, to make any type of parsing software work with corpus data, customized data preprocessing has to be performed on a per-corpus basis, with different algorithms for every corpus. The amount of effort needed to make one of these custom preprocessors vary as much as the corpora themselves as some high-quality ones (especially the proprietary ones) are very standardized, while yet other free ones are littered with errors and change internal notation every paragraph, or don't have any notation at all. The corpus used for this project (The Tanaka corpus) falls somewhere in-between of these two extremes. The preprocessing mainly focused on singling sentences out, and replacing the *kanji-kana* variants indicated by curly braces with a single type of notation (consecutive pipe characters for each variation). Other preprocessing operations included removing ~ characters and square brackets used by the WWWJDIC server, as these hold no interesting semantic information usable by the word prediction prototype. One interesting feature of these preprocessors that it would be very easy to implement several preprocessors for different kinds of corpora that could plug in to the same loading pipeline of the program (this is another perk of using special preprocess passes). To implement this, one would simply have to use a file-tag in each corpus to identify the type, or perhaps use a file naming convention to differentiate them, and then to just write some code to read the tag, and run the file through the appropriate algorithm.

The second part of preparation is the loading pipeline. If the preprocessors did their job correctly, the loading pipeline should be completely corpus agnostic and should simply be able to linearly run through the processed corpus to consume its tokens. As hinted in the previous sentence; the main responsibility of the loader should be to crawl each processed corpus, word by word, analyzing the relationships found in each sentence, which in the case of this implementation only extends to bigram relation counting and particles. It then loads these relationships into the back end database. The interface between the loader and the database was also designed to be as decoupled as possible, so if someone would like to change the database module, they would do good to start looking in the `LoadObjectDatabases` method.

4.3 Data structures and plumbing

All the preprocessing and loading in the world won't do us any good without a solid back end database to store and organize our gathered data into objects and relations. The current prototype has two separate data containers that together make up its back end database. The first container is called the `LexRefDatabase` and the other the `LexObjDatabase`. The two containers have distinct responsibilities and are orthogonally designed; any change in one container should not necessarily have to impact the data in the other container. The `LexRefDatabase` handles all relations in the database, storing bigram data and an interface to access them in O(1)[16] time. The database is structured as a data dictionary[17] where the keys are strings which represent the first word in a particular bigram, and the values are objects wrapping the possible second words in the bigram sequence; so called `LexRefContainer` objects. Each `LexRefContainer` object maintains a list of `LexicalReference` objects, which are the core components of the `LexRefDatabase`. Each `LexicalReference` embodies a potential followup word, as well as its popularity. The popularity is simply the number of occurrences of the key/value bigram pair.
The `LexObjDatabase` on the other hand, is responsible for keeping the closer details for each word, such as variations of the word and it's most common prepositions. This object was designed for extensibility as more heuristics come into play, and is also implemented using a data dictionary. The prepositions are mostly particles and are therefore embodied by the `Particle` class. The `Particle` class is very similar to the `LexicalReference` class in that it acts as a wrapper for the particle name and its popularity. One difference is that the particle class also contains static properties for parsing particles into a more manageable data format.
The so called "plumbing" of a software is an expression for how the different components in the system fit together. These relationships are best expressed with a dependency graph. The following graph was generated directly inside Visual Studio, with the different colors indicating tier and the arrows indicating dependencies. The thickness of each arrow indicates the degree of dependency, so the main WordPredictor class is in other words heavily dependent on the `LexRefContainer` and `LexicalObject` classes, but not as much on the `LexicalReference` and `Particle` classes.



The prototype class diagram showing the dependencies of each component. In addition to these concrete classes, the prototype also contains abstract components like the databases.

4.4 Input processing and information retrieval

The last internal component of the prototype is the input processing and information retrieval pipeline. This is the primary conduit by which the user interacts with the underlying data model. The input box located on the graphical user interface is hooked up to an event handler that detects any changes in the text box. When a space is detected, a parse request is sent to examine the content of the text box. The content is tokenized and the last token is used to send a request to the `LexRefDatabase`. The `LexRefContainer` returned by the request is then examined and the *n* most popular `LexicalReference`s are returned to be processed. Each reference is then sent to the `LexObjDatabase` to retrieve its corresponding `LexicalObject`. The returned objects are all mined for their most popular particles and any available word variations. All this information is then outputted to the suggestion matrix below the input box.

4.5 Performance vs. accuracy

In a full edition word prediction engine, the classic computer science problem of performance vs. accuracy comes into play. Since the prototype is devoid of most heuristics and the only computation-heavy operation is the database load, this is hard to demonstrate without using proprietary software. However, the problem basically boils down to deciding whether CPU-intensive heuristics should be run or not to improve suggestion accuracy. Most heuristics are variations on exhaustive searches or Viterbi algorithms with runtime complexity touching on $O(n^2)$ or worse. This leads to scenarios where the program might have to decide during runtime if, and which, heuristic should be run. And if it is run, whether additional heuristics should be applied to the heuristic itself to perhaps limit its maximum allowed running time, or to cut down its search space.

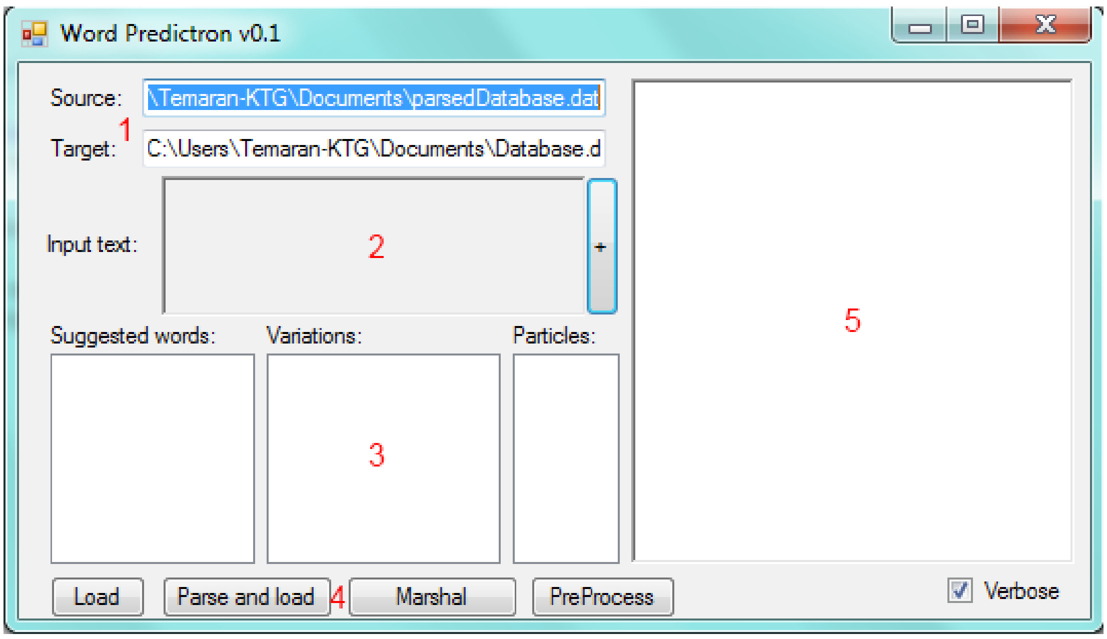
4.6 Data marshaling

It is recommended to load the prototype by using the "Parse and load" button at present. This however is ridiculously slow and unfit for real world use. To improve this load time, it would be effective to skip the loading parse logic completely and simply serialize[18] an already loaded model to later be able to deserialize it into memory again. There are a few problems with this, the first one being that using the built-in serialization mechanism in the .net framework is unacceptably slow since it serializes *all* data (object and overhead as well) in the model. It would be more efficient to write a custom serializer to only marshal the relevant data. Code would also have to be written to reverse the process. This will be explored a bit further in the **6.7 Serialization engine** chapter.

4.7 User interface

The user interface provides the means to the user to interact with the software itself. Thanks to the tools present in the .net framework, the software was developed with a graphical user interface from the offset. The interface consists of 5 main components:

1. The **Path control boxes**. Input and output paths are defined here.
2. The **Input area**. Text is inputted here to be used as data for the processing pipeline. Space initiates a parse.
3. The **Suggestion matrix**. Each column is a suggestion aspect while each row contains different suggestions sorted in descending order by popularity.
4. The **Action buttons**.
   1. The ***Load*** button loads a previously marshaled database into memory from the *Source* path. Currently *does not work as intended* as the marshaling algorithm is not optimized whatsoever.
   2. The ***Parse and load*** button first parses the *Source* file path and then loads it into memory. This is currently the recommended way to load the software.
   3. The ***Marshal*** button marshals the currently loaded database into the specified *Target* path. Currently *does not work as intended* as the marshaling algorithm is not optimized whatsoever.
   4. The ***Preprocess*** button preprocesses the *Source* path into the *Target* path. Currently only works with the complete version of the Tanaka corpus.
5. The **Feedback area** lists runtime feedback from the program such as parsing or loading process and errors. The **Verbose** checkbox can be checked to provide even more extensive information.


Source: \Temaran-KTG\Documents\parsedDatabase.dat
Target: C:\Users\Temaran-KTG\Documents\Database.d

The graphical user interface of the prototype.

5. Conclusions

While there still remain many things to be done before the prototype included in this report can be used as an assistive tool, it seems safe to say that word prediction is plausible even in a language as different from English as Japanese. Most of the problems faced when implementing this type of software in Western languages seem to stay the same when applied to their Eastern counterparts, such as the parsing methods, data structures and heuristics, even though the heuristics part remains unproven. One additional unexpected insight is that multipass conjugation pruning seems applicable even in Japanese. This needs further testing before it is proven however.
Japanese did however pose a few additional problems; mostly in the context of input conversion and the *wakachigaki* problem. There did not seem to be any parts of the implementation that actually got *easier* in the Japanese implementation compared to a Western language.
I do however think that word prediction software running on Eastern languages have the *potential* to become more accurate than Western systems simply because of the unambiguous nature of logograms. In the case of Japanese, this of course requires that *kanji* is used as input, as the alternative (*kana* / roman letters) are most likely *less* effective than Western systems because of *wakachigaki* problems. These statements about the relative ambiguity of Western writing systems compared to Eastern ones are all speculation however, and require further scientific inquiry before any definitive conclusions can be drawn.

6. Further avenues of research

I tried to summarize all potential improvements that I could think of, both for the prototype itself, but also to the general theory as well as some anecdotes.

6.1 UI

The user interface of the application could be greatly improved in many different ways. The most obvious improvement might be to port it to a GUI more similar to the one found in SonoScribe, since the primary users of the software would be unable to use the current UI at all! For this to be done however, it would be necessary to move the application core from the winforms platform entirely. Maybe using the new Windows Presentation Foundation (WPF) would be the best choice as it is almost as easy to implement compared to winforms, but has much better styling support. The feedback window is obviously also superfluous in a more application-oriented approach.

6.2 *Henkan* engine or interface

One of the biggest flaws in the current prototype is that it only supports parsing of *kanji* compounds and *kanji*-stemmed native words. As such, adding *kana* and roman letter support would be warranted. This would probably be harder than it might seem though since it would require changes in all levels of the application, unless some type of IME API[19] could be leveraged to simply convert inputted roman letters and *kana* into their *kanji* counterparts behind the scenes! This would at least add partial *kana*-support but would have several inadequacies. It would for example not parse *kana*/roman letters that do not have a *kanji* representation, and would not have any heuristics pertaining to which suggestion is chosen. This could of course be handled on the applications side, perhaps by utilizing the `LexRefDatabase` and `LexObjDatabase` to determine which of the suggested tokens is the most popular in the current context.

6.3 Heuristic improvements

As mentioned before; one interesting aspect of using *kanji* is the possibility to lower the overall token ambiguity in the system. There doesn't seem to be a whole lot of research on this subject, which seems strange considering the implications. If for example written Japanese is easier to semantically parse than say written English, it might be warranted to do directed experiments on automated language acquisition systems on idiomatic languages first, before trying them on “harder” languages. I'm very interested in finding any research at all on this subject, so if any reader knows of any, I would be most grateful if you could contact me about this by mail.

6.4 Support for other corpora

As discussed in the beginning of the thesis, the corpus used was not really optimized for this type of experiment, so using another base corpus might have been a good idea. An even better solution, however, would be to simply extend the program by adding preprocessors for more corpora. More underlying data usually means better accuracy, so this will always remain as a way to continually improve the software.

6.5 Word variation handling

The way the prototype deals with morphological parts of speech is at the moment sub-par and should be improved. This applies both to the parsing aspect as well as the suggestion aspect. Dealing with morphology in word prediction software is among the hardest problems present though since there are few clear indications in the leading text on which conjugation the user intends to use. Even advanced software like SonoScribe approaches the problem more or less naively due to a lack of effective heuristics. It is still better than the prototype's solution though. As described earlier in the thesis, the prototype just saves complete lumps of conjugation morphology with no semantic analysis whatsoever. As the morphological markers in Japanese have many of the characteristics common to what linguists call "closed" word classes, which means that there is only a finite amount of conjugations available (you cannot make up new ones on the fly) it would be better to at least use some technique, for example multipass pruning, or maybe even some simple *wakachigaki*-algorithm to tokenize these lumps into a set of known symbols. These tokens would then be treated much like the `Particle` class in that each morpheme would be treated as a "word", and the suggestion matrix's second column would consequently be populated with these single morphemes instead. To accommodate for the agglutinative nature of Japanese, one could make sure that after completing a suggestion of a word where a morpheme was used, the next suggested word would not be a stem, but another possible morpheme until the most popular followup morpheme is null. This would lead to a much more dynamic morphological suggestion system, and once again, since we could treat the morphemes as a closed word class, it would also likely save us a lot of memory usage.

6.6 The input *wakachigaki* problem

The *wakachigaki* problem has been touched on several times in this thesis already, but there is still some things that could be improved on. So far we've only really talked about the *wakachigaki* problem in the context of text and corpus parsing. With this paragraph, I just wanted to make the reader aware of the other, less obvious *wakachigaki input problem*. The reader should be familiar with the fact that written Japanese lacks delimiter characters, and that this impairs computer based parsing by now. This applies to *any* text the software has to parse, including the input string. Consider the following example sentence:

次	の	言葉	は	何
Tsugi	no	kotoba	ha	nani
Next	poss.	word	topic.	what

This sentence would be written into the input area as: 次の言葉はなに
The problem is now;
How is the program supposed to know which part of this sentence constitutes its "last word", i.e. which part of the sentence should be used as the candidate string for the `LexRefDatabase`? The answer is of course to use *wakachigaki*!
The next problem is then;
How do we let the computer know that we want a suggestion procedure to take place? The best way in a keyboard oriented software would probably be to just assign a key to be the "suggestion key". However, the ideal GUI for this program doesn't have any buttons! This means that we either have to introduce some kind of other delimiter into the system, which would defeat much of the point of the system by slowing it down. Or, we could simply prompt the suggestion matrix on *any* input. This of course, would be rather slow, but still preferable to the alternative. It is, in other words, yet again an accuracy vs. performance type of problem we are dealing with.
The prototype itself, however, uses neither approach as it does not employ the desired GUI yet. To simplify development, space is simply used as the delimiter that prompts a suggestion. This should be changed as soon as there is time.

6.7 Serialization engine

To be viable as a truly usable software, the database loading time has to be sped up considerably. The best way to achieve this would be with the use of a dedicated serialization/deserialization engine that when prompted can serialize only the most relevant data in each object onto disk. It should then be able to reverse the process by recreating all objects and loading them with the stored data.

6.8 Analyzing the performance of the prototype

If I had been able to put a bit more work into the prototype itself, making a performance analysis of the program would have been desirable to see if the number of keystrokes saved in Japanese is comparable to that of English. There is no real point in doing one at this point in time though for several reasons. One of the reasons and perhaps the most obvious one is that it as a prototype lacks almost all semantic heuristics present in a modern proprietary engine. Another reason is the lack of time. I do however aim to continually improve the software so that it might one day be worthy to be tested against others of its kind. And since the source code will be made public, perhaps some other programmers interested in linguistics will help me with this endeavor to in the future make the Word Predictron a world class word prediction engine.

7. References and notes

[1]. There are no real dates that indicates “when the world became literate” but the last 200 years did a lot to Western literacy, and illiteracy halved in the last 30 years according to Wikipedia:
http://en.wikipedia.org/wiki/Literacy

[2]. e.g. “Hello, my name is X”

[3]. Keystrokes saved is a mark used in AAC research that is measured using the following formula:

$$\frac{keystrokes_{letter\text{-}by\text{-}letter} - keystrokes_{with\ prediction}}{keystrokes_{letter\text{-}by\text{-}letter}} \times 100\%$$

[4]. Data from internal Tobii Technology documents and:
http://www.cis.udel.edu/~trnka/research/trnka08evaluating-presentation.pdf

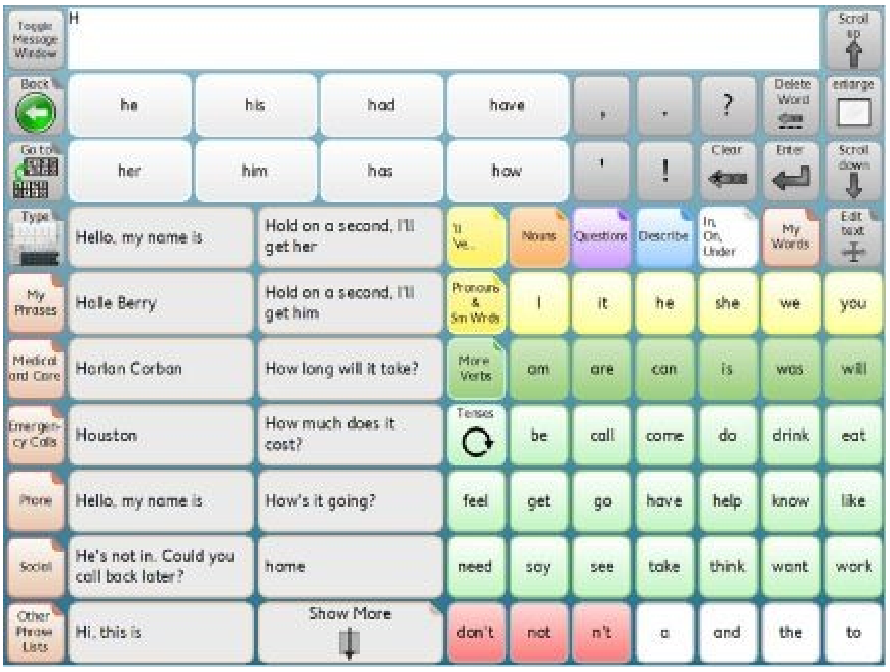
[5]. The FASTY project
http://www.elearningeuropa.info/directory/index.php?page=doc&doc_id=971&doclng=6

[6]. Market share info can for example be found here:
http://www.networkworld.com/community/blog/windows-drops-below-90-market-share

[7]. Part of speech tagging:
http://en.wikipedia.org/wiki/Part-of-speech_tagging

[8]. Conditional random fields:
http://en.wikipedia.org/wiki/Conditional_random_field

[9]. A screen shot of SonoScribe:



[10]. Wakachigaki is a very interesting problem not found in Western languages. Many research papers are as a consequence not available in English. My paper of choice is "Morphological analysis and wakachigaki processing" (形態素解析と分かち書き処理), by Yasuda Akio:
http://wordminer.comquest.co.jp/wmtips/pdf/H15_01-4.pdf

[11]. Bigrams are combinations of two word tokens used in statistical analysis of for example corpora.
http://en.wikipedia.org/wiki/Bigram

[12]. According to the DTP Informe network (Japanese):
http://www.informe.co.jp/useful/character/character20.html

[13]. The Viterbi algorithm is widely used in NLP to make statistical decisions on which the most likely sequence of tokens might be from a set of possible sequences.
http://en.wikipedia.org/wiki/Viterbi_algorithm

[14]. Many systems use this approach, one example would be Martin Hassel's stemming lab:
http://nlp.lacasahassel.net/stemminglab/

[15]. Coupling is a concept in software engineering to describe how much dependency exists between two components. A lot of decoupling is generally good since it allows for more maintainable code and less side-effects. When performed correctly, it can also allow a third party to switch entire modules as long as they respect the I/O conventions of the module.
http://en.wikipedia.org/wiki/Coupling_%28computer_science%29

[16]. Big O notation is a compact way to represent algorithm efficiency. O(1) indicates that an operation can be performed in constant time (as opposed to linear time or logarithmic or what have it). More information at:
http://en.wikipedia.org/wiki/Big_O_notation

[17]. Data dictionaries (the proper technical term is “associative array”) are popular data structures used for fast lookup of objects using so called “keys”. Refer to the following wiki-link for more information:
http://en.wikipedia.org/wiki/Associative_array

[18]. To read more about data serialization / marshaling, have a look at the following link:
http://en.wikipedia.org/wiki/Serialization

[19]. There actually exists a good API for this!
http://technet.microsoft.com/en-us/library/ms970191.aspx

8. Bibliography

8.1 Reference Literature (used when designing the implementation)

Tobii technology communicator suite documentation (SonoScribe)

Analysing performance in a word prediction system with multiple prediction methods [An article from: Computer Speech & Language]
(2007) by P.A. Vayrynen, K. Noponen, and T. Seppanen

The handbook of japanese linguistics
(1999) by Natsuko Tsujimura [Blackwell publishing]

An introduction to japanese linguistics 2nd ed.
(2007) by Natsuko Tsujimura [Blackwell publishing]

Speech and language processing 2nd ed.
(2009) by Daniel Jurafsky and James H. Martin [Pearson Education]

Data-Driven Syntactic Analysis Methods and Applications for Swedish
(2002) by Beáta Megyesi [Centre for speech technology, KTH, Stockholm]

8.2 Internet links (2011-04)

English:
http://www.wikipedia.org
http://www.cis.udel.edu/~trnka/research/trnka08evaluating-presentation.pdf
http://www.networkworld.com/community/blog/windows-drops-below-90-market-share
http://nlp.lacasahassel.net/stemminglab/
http://technet.microsoft.com/en-us/library/ms970191.aspx

Japanese:
http://mecab.sourceforge.net/
http://www.informe.co.jp/useful/character/character20.html
http://wordminer.comquest.co.jp/wmtips/pdf/H15_01-4.pdf

8.3 Corpus
http://www.edrdg.org/wiki/index.php/Tanaka_Corpus
http://www.tokuteicorpus.jp/