CHAPTER 1

INTRODUCTION

Natural language processing (NLP) is a field of computer science, artificial intelligence, and linguistics concerned with the interactions between computers and human (natural) languages. As such, NLP is related to the area of human–computer interaction. Many challenges in NLP involve natural language understanding, that is, enabling computers to derive meaning from human or natural language input, and others involve natural language generation.

The history of NLP generally starts in the 1950s, although work can be found from earlier periods. In 1950, Alan Turing published an article titled "Computing Machinery and Intelligence" which proposed what is now called the Turing test as a criterion of intelligence. NLP began in the 1950s as the intersection of artificial intelligence and linguistics. NLP was originally distinct from text information retrieval (IR), which employs highly scalable statistics-based techniques to index and search large volumes of text efficiently: Manning *et al*[1] provide an excellent introduction to IR. With time, however, NLP and IR have converged somewhat. Currently, NLP borrows from several, very diverse fields, requiring today's NLP researchers and developers to broaden their mental knowledge-based significantly.

Early simplistic approaches, for example, word-for-word Russian-to-English machine translation, were defeated by *homographs*—identically spelled words with multiple meanings—and metaphor, leading to the apocryphal story of the Biblical, ‘the spirit is willing, but the flesh is weak’ being translated to ‘the vodka is agreeable, but the meat is spoiled.’

Chomsky's 1956 theoretical analysis of language grammars provided an estimate of the problem's difficulty, influencing the creation (1963) of Backus-Naur Form (BNF) notation. BNF is used to specify a ‘context-free grammar’ (CFG), and is commonly used to represent programming-language syntax. A language's BNF specification is a set of *derivation rules* that collectively validate program code syntactically. (‘Rules’ here are absolute constraints, not expert systems' heuristics.) Chomsky also identified still more restrictive ‘regular’ grammars, the basis of the *regular expressions* used to specify text-search patterns. Regular expression syntax, defined by Kleene (1956), was first supported by Ken Thompson's *grep* utility on UNIX.

Subsequently (1970s), lexical-analyzer (lexer) generators and parser generators such as the *lex/yacc* combination utilized grammars. A lexer transforms text into tokens; a parser validates a token sequence. Lexer/parser generators simplify programming-language implementation greatly by taking regular-expression and BNF specifications, respectively, as input, and generating code and lookup tables that determine lexing/parsing decisions.

While CFGs are theoretically inadequate for natural language, they are often employed for NLP in practice. Programming languages are typically designed deliberately with a restrictive CFG variant, an LALR (1) grammar (LALR, Look-Ahead parser with Left-to-right processing and Rightmost (bottom-up) derivation), to simplify implementation. An LALR (1) parser scans text *left-to-right*, operates *bottom-up* (ie, it builds compound constructs from simpler ones), and uses a *look-ahead* of a *single* token to make parsing decisions.

The Prolog language was originally invented (1970) for NLP applications. Its syntax is especially suited for writing grammars, although, in the easiest implementation mode (*top-down* parsing), rules must be phrased differently (ie, right-recursively) from those intended for a *yacc*-style parser. Top-down parsers are easier to implement than bottom-up parsers (they don't need generators), but are much slower.

The limitations of hand-written rules: the rise of statistical NLP

Natural language's vastly large size, unrestrictive nature, and ambiguity led to two problems when using standard parsing approaches that relied purely on symbolic, hand-crafted rules:

- NLP must ultimately extract meaning (‘semantics’) from text: formal grammars that specify relationship between text units—parts of speech such as nouns, verbs, and adjectives—address syntax primarily. One can extend grammars to address natural-language semantics by greatly expanding sub-categorization, with additional rules/constraints (eg, ‘eat’ applies only to ingestible-item nouns). Unfortunately, the rules may now become unmanageably numerous, often interacting unpredictably, with more frequent *ambiguous parses* (multiple interpretations of a word sequence are possible). (Puns—ambiguous parses used for humorous effect—antedate NLP.)

The 1980s resulted in a fundamental reorientation, summarized by Klein:

- Simple, robust approximations replaced deep analysis.
- Evaluation became more rigorous.
- Machine-learning methods that used probabilities became prominent. (Chomsky's book, *Syntactic Structures*–(1959) had been sceptical about the usefulness of probabilistic language models).
- Large, annotated bodies of text (corpora) were employed to train machine-learning algorithms—the annotation contains the correct answers—and provided gold standards for evaluation.

This reorientation resulted in the birth of *statistical NLP*. For example, *statistical parsing* addresses parsing-rule proliferation through probabilistic CFGs:individual rules have associated probabilities, determined through machine-learning on annotated corpora. Thus, fewer, broader rules replace numerous detailed rules, with statistical-frequency information looked up to disambiguate. Other approaches build probabilistic 'rules' from annotated data similar to machine-learning algorithms like C4.5,-which build decision trees from feature-vector data. In any case, a statistical parser determines the *most likely* parse of a sentence/phrase. 'Most likely' is context-dependent: for example, the Stanford Statistical Parser rained with the Penn TreeBank annotated *Wall Street Journal* articles, plus telephone-operator conversations—may be unreliable for clinical text. Manning and Scheutze's text provides an excellent introduction to statistical NLP.

Statistical approaches give good results in practice simply because, by learning with copious real data, they utilize the most common cases: the more abundant and representative the data, the better they get. They also degrade more gracefully with unfamiliar/erroneous input. This issue's articles make clear, however, that handwritten-rule-based and statistical approaches are complementary.

Major tasks in NLP

The following is a list of some of the most commonly researched tasks in NLP. Note that some of these tasks have direct real-world applications, while others more commonly serve as sub-tasks that are used to aid in solving larger tasks. What distinguishes these tasks from other potential and actual NLP tasks is not only the volume of research devoted to them but the fact that for each one there is typically a well-defined problem setting, a standard metric for evaluating the task, standard corpora on which the task can be evaluated, and competitions devoted to the specific task.

Automatic summarization

Produce a readable summary of a chunk of text. Often used to provide summaries of text of a known type, such as articles in the financial section of a newspaper.

Coreference resolution

Given a sentence or larger chunk of text, determine which words ("mentions") refer to the same objects ("entities"). Anaphora resolution is a specific example of this task, and is specifically concerned with matching up pronouns with the nouns or names that they refer to. The more general task of coreference resolution also includes identifying so-called "bridging relationships" involving referring expressions. For example, in a sentence such as "He entered John's house through the front door", "the front door" is a referring expression and the bridging relationship to be identified is the fact that the door being referred to is the front door of John's house (rather than of some other structure that might also be referred to).

Discourse analysis

This rubric includes a number of related tasks. One task is identifying the discourse structure of connected text, i.e. the nature of the discourse relationships between sentences (e.g.

elaboration, explanation, contrast). Another possible task is recognizing and classifying the speech acts in a chunk of text (e.g. yes-no question, content, statement, assertion, etc.).

Machine translation

Automatically translate text from one human language to another. This is one of the most difficult problems, and is a member of a class of problems colloquially termed "AI-complete", i.e. requiring all of the different types of knowledge that humans possess (grammar, semantics, facts about the real world, etc.) in order to solve properly.

Morphological segmentation

Separate words into individual morphemes and identify the class of the morphemes. The difficulty of this task depends greatly on the complexity of the morphology (i.e. the structure of words) of the language being considered. English has fairly simple morphology, especially inflectional morphology, and thus it is often possible to ignore this task entirely and simply model all possible forms of a word (e.g. "open, opens, opened, opening") as separate words. In languages such as Turkish, however, such an approach is not possible, as each dictionary entry has thousands of possible word forms.

Named entity recognition (NER)

Given a stream of text, determine which items in the text map to proper names, such as people or places, and what the type of each such name is (e.g. person, location, organization). Note that, although capitalization can aid in recognizing named entities in languages such as English, this information cannot aid in determining the type of named entity, and in any case is often inaccurate or insufficient. For example, the first word of a sentence is also capitalized, and named entities often span several words, only some of which are capitalized. Furthermore, many other languages in non-Western scripts (e.g. Chinese or Arabic) do not have any capitalization at all, and even languages with capitalization may not consistently use it to distinguish names. For example, German capitalizes all nouns, regardless of whether they refer to names, and French and Spanish do not capitalize names that serve as adjectives.

Natural language generation

Convert information from computer databases into readable human language.

Optical character recognition (OCR)

Given an image representing printed text, determine the corresponding text.

Part-of-speech tagging

Given a sentence, determine the part of speech for each word. Many words, especially common ones, can serve as multiple parts of speech. For example, "book" can be a noun ("the book on the table") or verb ("to book a flight"); "set" can be a noun, verb or adjective; and "out" can be any of at least five different parts of speech. Some languages have more such ambiguity than others. Languages with little inflectional morphology, such as English are particularly prone to such ambiguity. Chinese is prone to such ambiguity because it is a tonal language during verbalization. Such inflection is not readily conveyed via the entities employed within the orthography to convey intended meaning.

Parsing

Determine the parse tree (grammatical analysis) of a given sentence. The grammar for natural languages is ambiguous and typical sentences have multiple possible analyses. In fact, perhaps surprisingly, for a typical sentence there may be thousands of potential parses (most of which will seem completely nonsensical to a human).

Sentence breaking (also known as sentence boundary disambiguation)

Given a chunk of text, find the sentence boundaries. Sentence boundaries are often marked by periods or other punctuation marks, but these same characters can serve other purposes (e.g. marking abbreviations).

Sentiment analysis

Extract subjective information usually from a set of documents, often using online reviews to determine "polarity" about specific objects. It is especially useful for identifying trends of public opinion in the social media, for the purpose of marketing.

Topic segmentation and recognition

Given a chunk of text, separate it into segments each of which is devoted to a topic, and identify the topic of the segment.

Word segmentation

Separate a chunk of continuous text into separate words. For a language like English, this is fairly trivial, since words are usually separated by spaces. However, some written languages like Chinese, Japanese and Thai do not mark word boundaries in such a fashion, and in those languages text segmentation is a significant task requiring knowledge of the vocabulary and morphology of words in the language.

Word sense disambiguation

Many words have more than one meaning; we have to select the meaning which makes the most sense in context. For this problem, we are typically given a list of words and associated word senses, e.g. from a dictionary or from an online resource such as WordNet.

CHAPTER 2

PROBLEM STATEMENT

To implement: A live text editor that has the following:

1) ***Spell Checker/Corrector***: A live automated spell checker that detects and corrects misspelt errors. Underlines words that are not in the dictionary, option to add words into the dictionary:
    a. Non word error detection using a Hash Map amalgamation of a corpus.
    b. Generating candidates of the misspelt word, using Levenshtein's Edit Distance Model (incorporated an edit distance of 2).
    c. Automatic correction using a Bayesian Model and the Noisy channel Model to make the most appropriate choice of correction.
    d. Bayesian Model for word segmentation. For eg: 'iloveindia' to 'I love india'
    e. Add to dictionary.
    f. Underlining of words that are misspelt
    g. Right Click feature, to generate candidate words, if an error exists.
2) ***Word Segmentation***: A live feature that checks if the user has missed out spaces between words and tries to automatically add spaces.
3) ***Auto-complete***: An option that continuously scans for long words and completes them automatically whenever applicable.
4) ***Dynamic find***: That provides an option to the user to search for regular expressions
5) ***Topic Extraction***: To provide search links to the user to easily search for additional information on the text being typed. This requires text analysis, topic modeling, and keyword extraction.
    a. Detection using keyword mapping:
        i. Letter Detection
        ii. Resume Detection

        iii. Code Detection

    b. Topic/Keyword Extraction

        i. Essay theme

        ii. Word Frequency Analysis

6) ***Background Processing***: All features must be background processes, to enable seamless smooth operations of the editor.

7) ***An Editor Interface***: A neat GUI with simple design that houses all of the above features.

CHAPTER 3

SYSTEM REQUIREMENTS

Hardware and OS Requirements:

1. Windows 8 (Desktop)
2. Windows 7
3. Windows Vista SP2
4. Windows Server 2008
5. Windows Server 2012 (64-bit)
6. RAM: 128 MB; 64 MB for Windows XP (32-bit)
7. Disk space: 124 MB
8. Browsers: Internet Explorer 7.0 and above, Firefox 3.6 and above, Chrome

Java Requirements:

1. Java SE 6.0 or greater
2. JRE 1.6 or greater
3. JDK 6.0 or greater

CHAPTER 4

DESIGN

The design involves a slight modification of the famous MVC architecture pattern.

- **Text Area**

  A simple swing component that allows the user to type in content.

- **Listeners To The Text Area**

  Keyboard and mouse events, which are triggered automatically.

- **Stored Data**

  Which includes the corpus text files, serialized files.

- **Backend Algorithms**

  Powerful concepts of Natural Language Processing

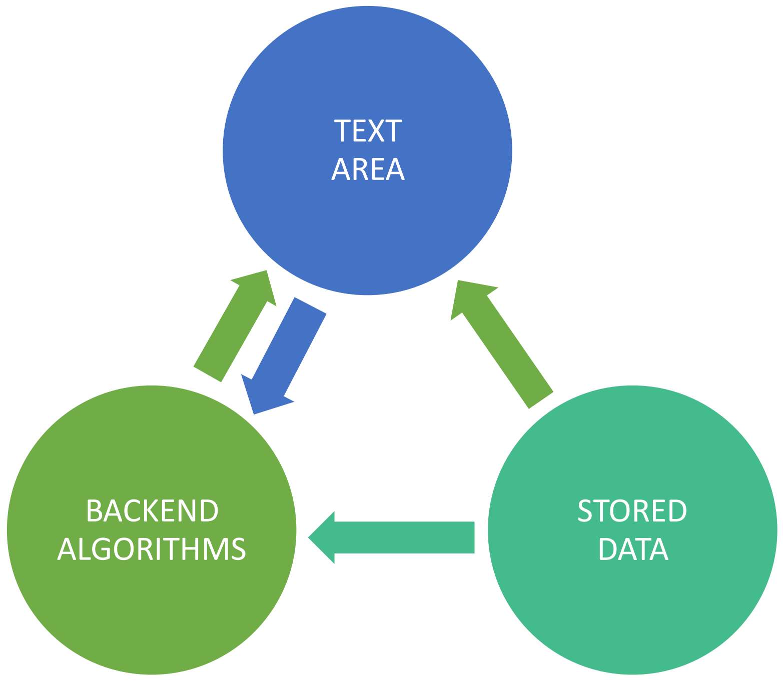


Figure 4.1: Flow Diagram of modified MVC

The Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user. The central component, the model, consists of application data, business rules, logic and functions. A view can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. The third part, the controller, accepts input and converts it to commands for the model or view.

Analysing the design of this architecture, the Integrated Text Editor is divided into the 3 MVC components, but with slight adjustments to suite the design.

1. The Text Area, that conglomerates the various options of the Editor, may be associated with the View Component of MVC. The TextArea contains active listeners that are triggered, each time, the user presses any input.
2. The Backend Algorithms may be associated with the Controller component of MVC, which reacts to the all the listeners, and performs the appropriate actions, and executions.
3. The Stored Data, which contains stored serialized hash map objects, corpus text files, may be associated with the Model Component of MVC.

CHAPTER 5

IMPLEMENTATION

5.1 Spell Checker/Corrector

A major functionality of the Integrated Text Editor is the detection and correction of spelling errors. Given a misspelt word, we would like our editor to detect the spelling error as well as replace it with the most viable candidate word. At the outset, let us first discuss the types of spelling errors that we may come across.

There are two types of spelling errors that we have to deal with:

1. Non word spelling error
2. Real word spelling error.

5.1.1 Non word spelling error

A spelling error that is not found in a dictionary is termed a non-word spelling error.

Eg: graffe for giraffe

5.1.2 Real word spelling error

A spelling error that is a valid word found in a dictionary is a real word spelling error. These spelling errors necessitate the inspection of the context in which the misspelling occurs for its detection and subsequent correction.

Eg: desert for dessert and vice versa

5.1.3 Non-word spelling error detection and correction

Obviously, detection of non-word spelling errors isn’t a very difficult task. With a sufficiently large and dependable corpus (dictionary), any word that is not found in the dictionary can be safely labelled as a non-word spelling error.

Correction however is slightly more involved. To understand how to implement an algorithm that corrects a non-word spelling error with a viable candidate, it would help to first examine how we humans are able to correct non-word spelling errors. The word graffe is obviously meant to be giraffe, but how did we figure this out? The fundamental reason for such intuition is our ability to identify associations between similarly spelt words. We hence have to quantify this intuition of association between words in order to develop an efficient non-word spelling error correction algorithm. In order to do this, we explore a quantity known as edit distance.

5.1.4 Edit distance

Edit distance is a metric to measure how dissimilar two words are to one another by counting the minimum number of operations required to transform one string into the other. Hence lesser the edit distance between two words, closer they are to each other in terms of our intuition of association. Common edit operations include insertion, deletion, substitution and transposition. These operations could be assigned different weights while computing the edit distance between two words. We however compute the edit distance after assigning equal weights to all the aforementioned edit operations namely the Levenshtein edit distance, where each operation has an equal weight of 1.

Eg:

The Levenshtein distance between "kitten" and "sitting" is 3.

1. kitten → sitten (substitution of "s" for "k")
2. sitten → sittin (substitution of "i" for "e")
3. sittin → sitting (insertion of "g" at the end).

We now have to generate all words that are a certain edit- distance away from a given target word (word recognized as a non-word spelling error). The literature on spelling correction claims that 80 to 95% of spelling errors are an edit distance of 1 from the target. We will also take into consideration words that are at an edit distance of 2 from the target word.

Hence upon detection of a non-word spelling error, we generate a set of valid words that are 1 or 2 edit-distances away from the target word. Now we have to pick the most 'viable' candidate word that belongs to this set of words. Hence there is a need to enumerate the different candidate words that belong to this set. This is where we introduce the noisy channel model for spelling correction, a model that is largely used for this task.

5.1.5 Noisy channel model for spelling correction

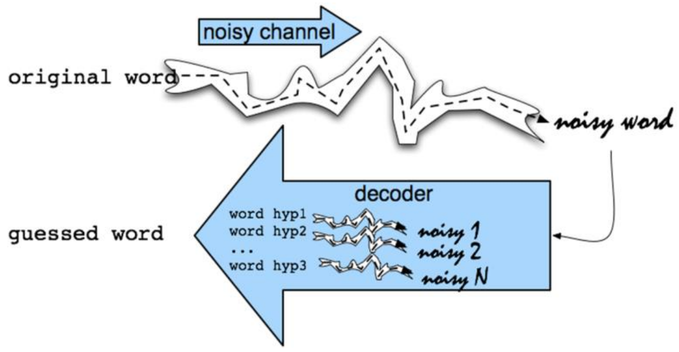


Figure 5.1: Noisy Channel

In this model, the goal is to find the intended word given a word where the letters have been scrambled in some manner (scrambled to form a misspelling in this case). Essentially we have a word *c* that is affected by noise (spelling error) in a noisy channel (the typist) to produce the misspelling *w*.

Our task is to correctly produce *c*, given *w*. This is done by applying conditional probability.

Formally, let *C* be the set of all candidate words identified given a misspelling w (this is done by identifying a set of all words that are at an edit distance of 1 or 2 away from *w*). Our task is to select $c \epsilon C$ such that *p(c|w)* is maximium.

The probability *p(c|w)* is given by :

$$p(c|w) = \frac{p(w|c).p(c)}{p(w)}$$

Since *p(w)* is the same for every possible *c*, we ignore it and obtain:

$p(c|w) = p(w|c).p(c)$………………………………………………………..Equation 2

We hence have to find $c \epsilon C$ such that *p(c|w)* given by equation 2 is maximized.

A look into each term in Equation 2 and its intuition will give us an idea of what this implies.

p(c|w)	Probability that the typist meant to type c, given he/she has typed w.
p(w|c)	Probability that a typist may type w when he/she meant to type c.
p(c)	Probability of occurrence of c in a corpus.

p(c|w) hence is directly proportional to *p(w|c)* and *p(c)*.

In our model we evaluate *p(w|c)* based on a simple assumption: lesser the edit distance between *c* and *w* where $c \epsilon C$, higher is *p(w|c)*. This means that if we find a $c \epsilon C$ that is an edit distance of 1 away from *w*, the corresponding *p(w|c)* is higher than that for those $c \epsilon C$ that are an edit distance of 2 from *w*. We then resolve conflict between those c ϵ C that are the same edit distance away from *w* by considering *p(c)*. The *c* with highest *p(c)* (having higher frequency in the corpus) is then chosen. The *p(c)* component makes sure that we don't suggest obscure words as corrections even if their edit distance from *w* happens to be the least.

This intuitively means, given a non-word spelling error, we choose the word with highest similarity in terms of spelling to the misspelt word, provided it is fairly common in our corpus.

Now that we have examined the principle behind our spelling error detection and correction mechanism, we can look at the finer implementation details.

General Methodology

1. Once a user types a word *w*, we search for the same within our corpus. If found, we don't do anything. If the word isn't found, we identify *w* as a non-word spelling error.
2. Once a typed word *w* is detected as a non-word spelling error, we generate *C*, the set of all words that are of an edit-distance of 1 or 2 from *w*.
3. We reduce *C* to contain only those words that are contained in the corpus.
4. We then choose $c \epsilon C$ such that its edit distance from *w* is least and its corresponding frequency of occurrence in the corpus is highest.

Flowchart

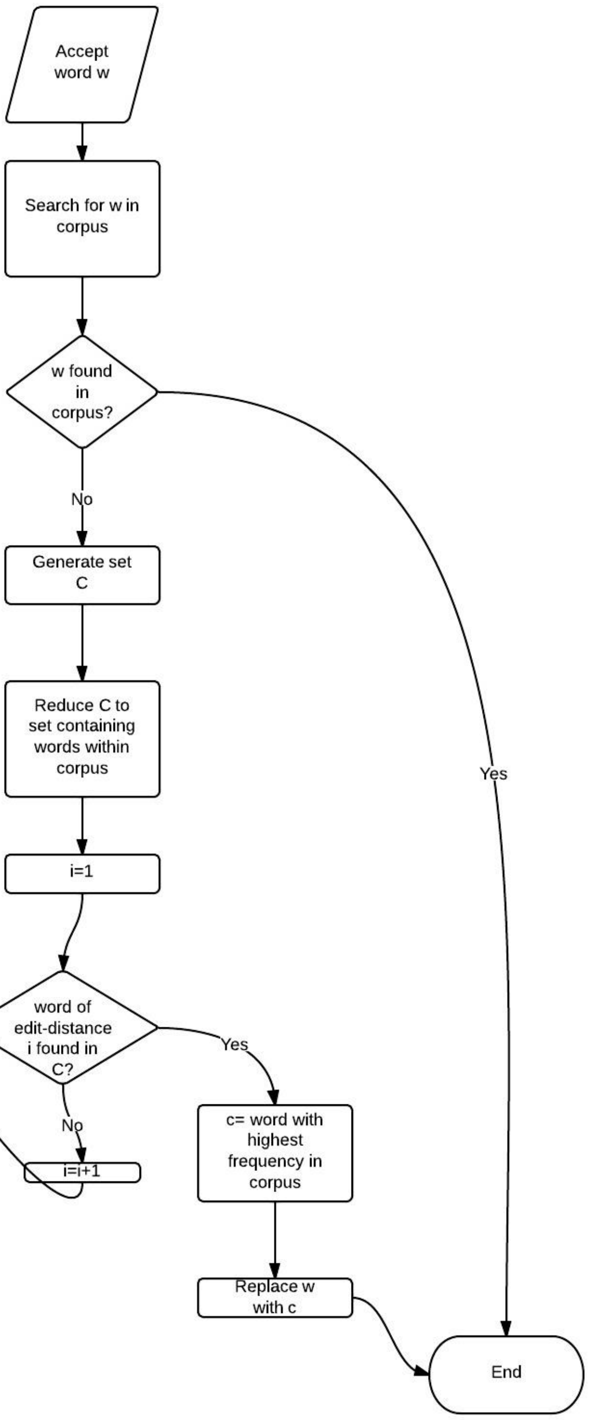

Accept word w
Search for w in corpus
w found in corpus?
No
Yes
Generate set C
Reduce C to set containing words within corpus
i=1
word of edit-distance i found in C?
Yes
No
i=i+1
c= word with highest frequency in corpus
Replace w with c
End

Figure 5.2: Spell Check data flow diagram

5.2 Word Segmentation

Apart from spelling errors discussed under non-word spelling errors, another major error typists are often prone to is the omission of space between different words. For example the sequence of words “Hello there” may often be typed as “Hellothere”. Segmentation of such ‘invalid’ words into their constituent segments is word segmentation. Word segmentation would be a very helpful feature in text editors. This function of segmentation is a part of our Spell check module.

5.2.1 Methodology

Given a word w of length c, the total number of segmentations possible is 2^{c-1}. Our task is to segment w in such a way that all or most of the constituent segments are valid words found in a corpus. In other words, the joint probability of occurrence of the constituent segments is to be maximized. Formally, if a word *w* is segmented into $w_1, w_2 \ldots . w_n$ the value f=$p(w_1)p(w_2)p(w_3) \ldots p(w_n)$ is to be maximized. Clearly this means that we assume the probability of occurrence of words in a corpus is independent of each other. We use the same corpus and dictionary we built for the purpose of non-word spelling error correction to compute the aforementioned values

There may also exist cases where one of the segments in a result may not occur in our corpus, assigning a probability of 0 to such a segment is not a good choice. For example consider *w*= ‘bbcamerica’ clearly we have to split w into the segments ‘bbc’ and ‘america’. In all likeliness the word *bbc* may not appear in our corpus and assigning a probability of 0 to such a segment would mean *f*=0 even though the segmentation is right. We hence had to wisely choose a value that is to be returned as the probability of such a word that does not appear in our corpus. We also had to keep in mind the fact that longer an unknown word, lesser is its likeliness of being a valid word. Keeping these conditions in mind we computed the probability of words that do not occur in the corpus as $\frac{1}{N\,.\,10^{l-2}}$ where *N* is the total number of words in our corpus and *l* is the length of the unfound word.

We defined two functions to help us in performing word segmentation:

1. *wordSequenceFitness(l)*- A function that returns f for a list l comprising of $w_1, w_2 \ldots . w_n$. For better scaling, the logarithm of probability each word is computed.
2. *Segment(word)* this function works on the basis of induction and returns the best segmentation of word.

As mentioned before *segment(word)* works on the basis of induction. In particular, we are working by induction on the length of a word. Assuming we know the optimal segmentations for all substrings not including the first letter, we can construct the best segmentation which includes the first letter. We look at all possible split pairs, including the one which considers the entire word as a good segmentation, and we find the segmentations that have the highest return value with respect to *wordSequenceFitness*. By induction we know that *segment* returns the optimal segmentation on every call, since each recursive call operates on a strictly smaller substring which does not include the first letter. Hence, we have covered all possible segmentations, and the algorithm is correct.

5.2.2 Steps of implementation:

1. We first define a function *segment* that produces all possible segmentations of a given word. This is easily implemented in Java using its easy to use substring functions for string.
2. We define *wordSequenceFitness* that operates on lists of words generated by *segment* and returns the combined probability of the constituent words. More precisely, logarithms of the probability values are returned in order to prevent generation of miniscule values.
3. Finally we define the function *segment* which contains the meat of the word segmentation task. Segment returns a list of words that correctly make up the argument (unsegmented word) of the function. This function is based on induction as mentioned above. This functioned is recursively called with substrings of the argument every time. The function returns an empty list when the argument is an empty string.

5.3 Autocomplete

The Integrated Text Editor provides a handy Autocomplete feature that helps the user type faster by remembering long words that have already been typed. Such a feature is not provided any of the famous text processors. Let us first observe the motivation behinds the invention of such a feature.

5.3.1 Motivation

Often when writing a document about a particular topic, certain terms or names are always repeated. Consider the user who is typing on the following topic:

"The Chicago Architecture Foundation (CAF) is a nonprofit organization based in Chicago, Illinois, USA, whose mission is to inspire people to discover why architecture and design matters. It is well known for its public programs, most notably the docent-led architecture cruise on the Chicago River, and bus, walking, bike, Segway and 'L' train tours of the Chicago-area. Chicago Architecture Foundation is also known for its scale model of downtown Chicago.

The Chicago Architecture Foundation was founded in 1966 as the Chicago School of Architecture Foundation to save ……"

In the above extract, words like Architecture, and Foundation. It would be very helpful if a text editor can keep track of such long words as and when they are typed and suggest the same when the user is about to type the word again in the future. The Integrated text editor does just that through its Autocomplete feature.

5.3.2 Working

Building a Saved List:

A HashMap named *Completable* is maintained that keeps track of all valid words (words in the corpus dictionary) of a particular length (7 in this case) when typed for the first time. Scan the document continuously for words that exceed a specific threshold (7 letters). Every time a long word is typed, the word is added to a Completable List.

Continuous Listeners:

Suggestions are shown when the user has typed at least 3 character of a new word. Based on the word being typed, and the words present in the Completable List, a new list of AutoComplete words is generated based on the matches. This process is live and hence does not hamper the typist’s typing routine.

Popup Feature:

The list of AutoComplete suggestions, is shown just below the cursor location, as a popup. The user may use keys, or the mouse to select one of the suggestions in order to use it, and save time. Upon typing a word subsequently, once the typist has entered three letters, *Completable* is traversed to find any potential matches which are displayed in a pop up window below the word, if found.

5.4 Topic Extraction

Topic extraction is possibly the most unique feature offered by our text editor. The aim of this functionality is simple- to try and find out what the user is typing and subsequently provide useful links to the user.

Our text editor broadly classifies a document typed by a user into the following buckets:

1. Letter
2. Resume
3. Code
4. Essay

The methodology used in detecting the first three types of documents is the same. The detection of essays involves the function of keyword extraction or heading extraction.

5.4.1 Detecting letters, resumes and code

Letters, Resumes and Code in various programming languages have certain characteristic features or *keywords*. We maintain a set of keywords for each of the three types of documents- letter, resume and code. We then scan the document and maintain a list of words that appear in the document. Each document type is associated with a hit count. Whenever a word in the document uniquely matches a tag, the hit count of the appropriate document is increased by 1. If the hit count of a document is less than 3, it is ignored. The document with the highest hit count is detected as the document being typed. If all of the documents have their associated hit count, the document is then assumed to not fall under any of the three types. We then proceed to detecting the topic of the essay being typed.

5.4.2 Essay Topic detection

The task may be divided into two tasks:

1. Heading Extraction- Check for a heading format, within the Textarea.
2. Keyword Extraction- Word Frequency Analysis, which functions using a numerical statistic (tf-idf), to detect words that are key to the document.

If the document is found to not fall under any of the three document types (letter, resume, and code), we assume that the document being typed is an essay and try to detect the topic of the essay. We first scan the document for a heading; if it exists we extract the same and provide links to it. If no headings were detected, we proceed by performing keyword extraction.

5.4.3 Keyword Extraction

The keyword extraction methodology used in the text editor is a slight modification of the *tf-idf* method of obtaining the keyword, largely used in related areas of NLP. We define a function *f* that weighs the term positively for the number of times the term occurs within the document, while also weighting the term negatively relative to the corpus(Brown Corpus in this case). Formally:

$f = \frac{freq(t,d)}{freq(t,c)}$ where $freq(t,d)$ is the frequency of term *t* in the current document d and $freq(t,c)$ is the frequency of the term *t* in the corpus *c*. The terms are then arranged in decreasing order of their corresponding *f* values and the links to the top few keywords are provided.

5.4.4 Google Search Results

After realizing the type of document, or the heading of the document, or the top 4 keywords of the document, they are stored in a single query string. The Google Gson class (Json for Java) is used to perform a query search from an online google search API, and retrieve top 4 search results and their links.

Popup menu

The search results, are then populated on a temporary popup, which appears on the screen. User may choose to click on them, and will be redirected to that website.

5.5 Find

A feature that helps user to find characters or words in the text typed in the editor and highlights them. Allows user to search using **regular expressions.**

5.5.1 Regular expression

In theoretical computer science and formal language theory, a regular expression is a sequence of characters that forms a search pattern. Mainly for use in pattern matching with strings or string matching, i.e. “find and replace”-like operations.

Each character in the regular expression is either understood to be a metacharacter with its special meaning, or a regular character with its literal meaning. Together, they can be used to identify textual material of a given pattern, or process a number of instances of it that can vary from a precise equality to a very general similarity of the pattern. The pattern sequence itself is an expression that is a statement in a language designed specifically to represent prescribed targets in the most concise and flexible way to direct the automation of text processing of general text files, specific textual forms, or of random input strings.

A **metacharacter** is a character that has a special meaning (instead of a literal meaning) to a computer program, such as a shell interpreter or a regular expression engine. In regular expressions, there are 12 metacharacters that must always be preceded by a backslash, \, to be used inside of the expression: The opening square bracket [, the closing square bracket], the backslash \, the caret ^, the dollar sign $, the period or dot, the vertical bar or pipe symbol |, the question mark?, the asterisk or star *, the plus sign +, the opening round bracket (and the closing round bracket).

If you want to use any of these characters as a literal in a regex, you need to escape them with a backslash. If you want to match 1+1=2, the correct regex is 1\+1=2. Otherwise, the plus sign will have a special meaning.

META CHARACTER	DESCRIPTION
.	Matches any character
[ ]	Matches a single character that is contained within the brackets
[^]	Matches a single character that is not contained within the brackets.
^	Matches the starting position within the string
$	Matches the ending position of the string or the position just before a string-ending newline.
*****	Matches the preceding element zero or more times
?	Matches the preceding element zero or one time.
+	`Matches the preceding element one or more times
|	The choice (also known as alternation or set union) operator matches either the expression before or the expression after the operator
{M,N}	Matches the preceding element at least *m* and not more than *n* times
( )	Defines a marked sub expression

Table 5.1: Regular Expressions and their descriptions

Examples

- .at matches any three-character string ending with "at", including "hat", "cat", and "bat".
- [hc]at matches "hat" and "cat".
- [^b]at matches all strings matched by .at except "bat".
- [^hc]at matches all strings matched by .at other than "hat" and "cat".
- ^[hc]at matches "hat" and "cat", but only at the beginning of the string or line.
- [hc]at$ matches "hat" and "cat", but only at the end of the string or line.
- \[.\] matches any single character surrounded by "[" and "]" since the brackets are escaped.
- s.* matches any number of characters preceded by s, for example: "saw" and "seed".

5.5.2 Implementation

Using various in - built packages, classes and their utilities (functions) of java along with java swing components and event listeners.

Packages used

- `javax.swing      :  For swing components.`
- `javax.swing.text:  For swing text features.`
- `java.awt         :  Abstract window toolkit package, for creating UI.`
- `java.awt.event   :  For events and listeners.`
- `java.util.regex  :  Regex package`

Apart from these we make use of **Highlighter** interface, **Matcher** and **Pattern** classes.

- javax.swing.text.Highlighter
- java.util.regex.Matcher
- java.util.regex.Pattern

Finding position of the character or word

To find a specified character or word in the text, javax.util.regex package is used, which provides necessary classes and functions.

1. ***Pattern*****:** A compiled representation of regular expression.
   `public Pattern  compile( String ):` `create instance of the pattern.`
2. ***Matcher*****:** An engine that performs match operations on a character sequence by interpreting a pattern.
   `public Matcher matcher( String ):` `Used to match pattern again text.`
   `public Boolean find():` `returns true if match is found.`
   `public String group():` `returns input sequence matched.`
   `public int start():` `returns index of the match.`

Code Snippet

```
    Pattern p=Pattern.compile(str); //  str: pattern to be matched.

    String typo=t.getText();     // typo: text typed in the editor.

    Matcher m=p.matcher(typo);
```

To highlight the characters or words

To highlight the words found by the matcher, Highlighter interface utilities are used.

public interface Highlighter {

public Object addHighlight(int p0,int p1,Highlighter.HighlightPainter p): Used to highlight words.

public void removeAllHighlight(): Used to remove all highlights.

public Highlighter getHighlighter(): Used to get Highlighter.

Code Snippet

```
while(m.find()){

try {
h.addHighlight(m.start(),m.start()+m.group().length(),DefaultHighlighter.D
efaultPainter);

    } catch (BadLocationException e1) {

      e1.printStackTrace();

}
```

addHighlight() function throws an exception(BadLocationException).To catch the exception, the function is surrounded within try and catch block.

- Using *start()* function, the position of the word in the text area, which is then passed as a parameter to the function *addHighlight()* as shown in the snippet above.
- *addHighlight()*, highlights the word using default option(BLUE in colour).

5.5.3 Replace

An option that allows user to replace a word or a character (multiple occurrences) with another.

Implementation: Brute Force

In computer science, **brute-force search** or **exhaustive search**, also known as **generate and test**, is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement, some of the examples are;

1. Linear search
2. Selection sort
3. Bubble sort

Packages used

- ***java.util.regex.Matcher*:** Provides matching operations.
- ***java.util.regex.Pattern* :** Provide functions for creating compiled versions of pattern.

Code Snippet

```
str=(String)t1.getText();

h.removeAllHighlights();

Pattern p=Pattern.compile(str);

String typo=t.getText();

Matcher m=p.matcher(typo);

while(m.find()){
      ar[i]=m.start();     /* array ar[] holds position of the word or
character */

      System.out.println(ar[i]);

      j=i+1;                    // j holds length of the array

      i++; }
```

To replace with the specified word or character

To replace, we use brute force approach where in at each position of the word or character to be replaced in the text area, we collect the text before the position and after the position and store it. At the position, we replace the word or character with another (replacement).Later, we concatenate the entire text. The process is repeated at each position of the word or character to be replaced.

- To collect the text before and after the position we make use of ***substring(int index)*** function of **String,** with appropriate arguments.
- To concatenate and set it to text area we use ***setText(String)*** function of TextArea.

Code Snippet

```
for(i=0;i<j;i++){

      String before=
            t.getText().substring(0,ar[i]+i*(t2.getText().length()-
            t1.getText().length()));

      String after= t.getText().substring(ar[i]+i*(t2.getText().length()-

            t1.getText().length())+t1.getText().length());

      t.setText(before+t2.getText()+after);

}
```

Examples

substring(0,4): retrieves substring from position 0 to position 3.

substring(4): retrieves substring from position 4 to end of the string.

5.6 User Interface

The **user interface**, in the industrial design field of human–machine interaction, is the space where interaction between humans and machines occurs. The goal of this interaction is effective operation and control of the machine on the user's end, and feedback from the machine, which aids the operator in making operational decisions.

In computing, **graphical user interface** (**GUI)**, sometimes pronounced "gooey" or "gwee") is a type of user interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation, as opposed to text-based interfaces, typed command labels or text navigation. GUIs were introduced in reaction to the perceived steep learning curve of command-line interfaces (CLI), which require commands to be typed on the keyboard.

5.6.1 Design

GUI of Integrated text editor (ITE) is designed using various swing containers and components of java. GUI of ITE contains a text area, various buttons, text fields, menu bar, tool bar, icons. Along with this, the GUI also pops up various pop menus and dialogs on user actions.

1. ***Text area***: Allows user to enter text.
2. ***Menu bar:*** For save and edit options.
3. ***Tool bar:*** For cut, copy and paste options.
4. ***Find text field:*** For dynamically find a character or a word in the text.
5. ***Options button***: For various editor features.
6. ***Advance button***: For replacing a word or a character with another.

5.6.2 Text area

Designed using JTextArea of swing package. A JTextArea is a multi-line area that displays plain text. It is intended to be a lightweight component that provides source compatibility with the java.awt.TextArea class where it can reasonably do so.

public class JTextArea extends JComponent

This component has capabilities not found in the java.awt.TextArea class. The superclass should be consulted for additional capabilities. The java.awt.TextArea internally handles scrolling. JTextArea is different in that it doesn't manage scrolling, but implements the swing Scrollable interface. This allows it to be placed inside a JScrollPane if scrolling behavior is desired, and used directly if scrolling is not desired. The text entered by the user is obtained using appropriate listeners such as ***KeyListener***.

Every swing component is added to a frame (JFrame), a Frame is a top-level window with a title and a border. The size of the frame includes any area designated for the border. A frame, implemented as an instance of the JFrame class, is a window that has decorations such as a border, a title, and supports button components that close or iconify the window. Applications with a GUI usually include at least one frame.

Features incorporated in text area of ITE;

- Word wrapping at the end of the line.
- Automatic scroll.
- Automatic conversion of first letter of a sentence to capital letter.

5.6.3 Menu Bar

Designed using JMenuBar of swing package. JMenu objects are added to the menu bar to construct a menu. When the user selects a JMenu object, its associated JPopupMenu is displayed, allowing the user to select one of the JMenuItems on it.

```
public class JMenuBar extends JComponent
```

Menu bar of ITE has following features;

- ***File***: For opening a new text file (new), saving the current file (save), saving the current file with a different name (save as...).
- ***Edit:*** For editing the current text file like cut (Cut), copy (Copy) and paste (Paste) options.

The option selected from the menu is recognized using ***ItemListener*****.** Options (save, save as, open) selected in the menu causes appropriate dialogs to pop up at the centre of the screen using ***showXXXdialog ()*** function of ***JFileChooser.***

JFileChooser: FileChooser provides a simple mechanism for the user to choose a file. Provides a dialog to navigate a file system. File choosers provide a GUI for navigating the file system, and then either choosing a file or directory from a list, or entering the name of a file or directory.

```
public class JFileChooser extends JComponent implements Accessible
```

5.6.4 Tool Bar

Designed using ***JToolBar*** of swing package. ***JToolBar*** provides a component that is useful for displaying commonly used Actions or controls. A ***JToolBar*** is a container that groups several components — usually buttons with icons — into a row or column. Often, tool bars provide easy access to functionality that is also in menus.

```
        public class JToolBar extends JComponent
```

Tool bar of ITE has following features;

- ***New:*** For opening new text file.
- ***Open:*** For an existing opening a text file.
- ***Save:*** For saving the current file.
- ***Cut:*** A button for removing the text from text area.
- ***Copy:*** A button for copying the specified text.
- ***Paste:*** A button for pasting the copied text into text area.

These features are associated with appropriate icons in the tool bar.

5.6.5 Find text field

A text field is designed using JTextField of swing package. TextField is a lightweight component that allows the editing of a single line of text. JTextField is intended to be sourcecompatible with java.awt.TextField where it is reasonable to do so. This component has capabilities not found in the java.awt.TextField class. The superclass should be consulted for additional capabilities.

```
public class JTextField extends JComponent
```

The text entered in the field is obtained using ***KeyListener***. The specified word is **found** in the text area and **highlighted** dynamically.

Also, the ***Find*** text field has a focus when user presses ***Ctrl + F.***

5.6.6 Options

Option is a button designed using ***JButton*** of swing package. ***JButton*** is implementation of a "push" button. Buttons can be configured, and to some degree controlled, by Actions. Using an Action with a button has many benefits beyond directly configuring a button.

A JButton generates an action event, which can be listened by ***ActionListener.***

Further, **Option** button raises a dialog which provides user to select various features;

- ***Auto complete:*** Completes long words automatically.
- ***Spell check:*** Checks for spelling mistakes and corrects it.
- ***Context recognizer:*** Recognizes the context of what user is trying to type.
- ***Underline:*** Underlines spelling errors.

A dialog is designed using ***JDialog*** of swing package, the main class for creating a dialog window. You can use this class to create a custom dialog, or invoke the many class methods in ***JOptionPane*** to create a variety of standard dialogs. ***JDialog*** by default is associated with buttons which when hit causes the dialog to disappear.

The dialog has various swing components for different features like **radio buttons, check boxes, text fields, labels and separators.**

- ***Radio Button:*** JRadioButton,
- ***Check box:*** JCheckBox, for editor features.

- ***Separators:*** JSeparator, for implementing divide lines.
- ***Label:*** JLabel, for static texts like headings.
- ***Text field:*** JTextField

These components in the dialog are set to certain default values initially. Dialog by default is associated with two buttons (OK and Cancel),

- ***Cancel****:* Restores the values of dialog components to default values.
- ***Ok****:* Sets the values according to user's choice.

5.6.7 Advance

A button for providing a dialog, which allows user to enter a word and its substitute. Every occurrence of the specified word is replaced with the substitute.

Button is designed using ***JButton***, which generates an action event. Action listener recognizes the event (button pressed) and raises a dialog box at the centre of the screen.

The dialog contains following components:

- ***Find:*** A text field (***JTextField***), which accepts user input (word or character) for finding the word or character in the text area. The text entered in obtained using key listener dynamically. Every occurrence of the specified word or character is highlighted dynamically.
- ***Replace:*** A text field (***JTextField***), which accepts the substitute. The word or character specified in the find field is replaced with the substitute. The text entered in obtained using key listener dynamically. Every occurrence of the word or character is replaced with the substitute.
- ***Replace*** (replace) ***button*****:** A button, which initiates the replacing operation.

This dialog is associated with an" Ok "button which when hit causes the dialog to disappear.

5.7 Background Processing

5.7.1 Concurrency in Java

Careful use of concurrency is particularly important to the Swing programmer. A well-written Swing program uses concurrency to create a user interface that never "freezes" — the program is always responsive to user interaction, no matter what it is doing.

A Swing program deals with the following kinds of threads:

- *Initial threads*, the threads that execute initial application code. Only one such (main) will exist in the program. The most important jobs are to create a new `Runnable` Object. This object initializes the GUI.
- The *event dispatch thread*, where all event-handling code is executed. Most code that interacts with the Swing framework must also execute on this thread. They must be short tasks that finish quickly.
- *Worker threads*, also known as *background threads*, where time-consuming background tasks are executed.

The programmer does not need to provide code that explicitly creates these threads: they are provided by the runtime or the Swing framework. The programmer's job is to utilize these threads to create a responsive, maintainable Swing program.

Like any other program running on the Java platform, a Swing program can create additional threads and thread pools.

5.7.2 Java Worker Threads

Worker threads have their tasks that run on them are created using ***`javax.swing.SwingWorker`***. This class has many useful features, including communication and coordination between worker thread tasks and the tasks on other threads.

When a Swing program needs to execute a long-running task, it usually uses one of the *worker threads*, also known as the *background threads*. Each task running on a worker thread is represented by an instance of ***`javax.swing.SwingWorker`***. ***`SwingWorker`*** itself is an abstract class; you must define a subclass in order to create a ***`SwingWorker`*** object; anonymous inner classes are often useful for creating very simple ***`SwingWorker`*** objects.

`SwingWorker` provides a number of communication and control features:

- The ***`SwingWorker`*** subclass can define a method, ***`done(),`*** which is automatically invoked on the event dispatch thread when the background task is finished.
- ***`SwingWorker`*** implements ***`java.util.concurrent.Future`*****.** This interface allows the background task to provide a return value to the other thread. Other methods in this interface allow cancellation of the background task and discovering whether the background task has finished or been cancelled.
- The background task can provide intermediate results by invoking ***`SwingWorker.publish`***, causing ***`SwingWorker.process`*** to be invoked from the event dispatch thread.
- The background task can define bound properties. Changes to these properties trigger events, causing event-handling methods to be invoked on the event dispatch thread.
- All concrete subclasses of ***`SwingWorker`*** implement ***`doInBackground`***; implementation of `done` is optional. ***`doInBackground`*** is the function that is used to perform the new background task within a new thread.

5.7.3 Actual Implementation

Effective usage of the **SwingWorke**r class is of utmost importance. Since the user is constantly typing, the smooth functionality of the typing task would not be possible if, the various features of the editor, hog most of the resources, and hinder the typing task, of the flow of usage. In order to account for the smooth functioning, various time consuming tasks have to be rendered through a background process.

The task is divided into 2 parts:

1. ***UnderlineBackground:*** A class that underlines mispelt words. This requires continuous scanning of the document for misspelt words.

2. ***CRBackground:*** A class that continuouly tries to recognize the document and the text that is being type within it.

These two classes are triggered at regular intervals, and the processes are carried out accordingly, as a background process, without interfering with the main thread.

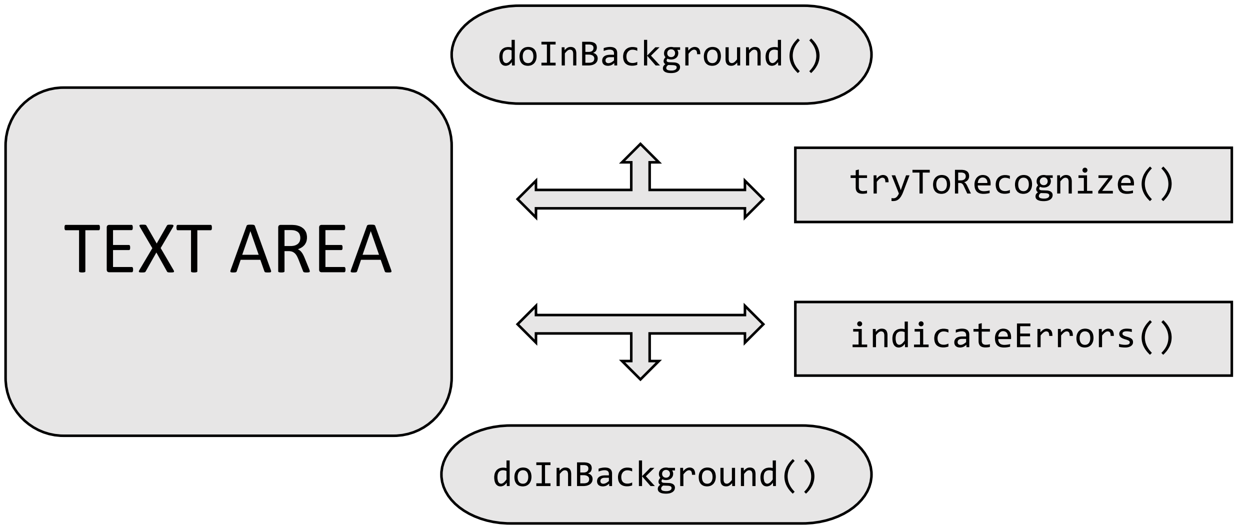


Figure 5.3: Functional Diagram of the Background processes

CHAPTER 6

RESULTS AND ANALYSIS

6.1 Testing

Software testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item. Software testing is an activity that should be done throughout the whole development process. Software testing is one of the “verification and validation,” or V&V, software practices.
There are two basic classes of software testing, ***black box testing and white box testing***.
Black box testing (also called functional testing) is testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.
White box testing (also called structural testing and glass box testing) is testing
that takes into account the internal mechanism of a system or component.

6.1.1 Three Types of Testing

1. ***Unit Testing** :* Unit testing is the testing of individual hardware or software units or groups of related units Using *white box testing techniques*, testers (usually the developers creating the code implementation) verify that the code does what it is intended to do at a very low structural level.

2. ***Integration Testing** :* Integration test is testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them Using both *black and white box testing techniques*, the tester (still usually the software developer) verifies that units work together when they are integrated into a larger code base.

3. ***System Testing**:* Functional testing involves ensuring that the functionality specified in the requirement specification works. System testing involves putting the new program in many different environments to ensure the program works in typical customer environments with

various versions and types of operating systems and/or applications. *System testing* is testing conducted on a complete, integrated system to evaluate the system compliance with its specified requirements.

6.2 Testing Results

Unit testing, integration testing and system testing were conducted to check for potential errors and correct them subsequently.

6.2.1 Unit Testing

ECLIPSE IDE was used to load and test the java code. *White box technique* was used to test individual class files for errors. Each code block has output print statements to verify the code for intended outputs. Eclipse console prints the output that helps to verify the correctness of the code as expected.

6.2.2 Integration Testing

ECLIPSE IDE was used to test the components after integration of sub-components which corresponds to different features of the text editor using both *white and black box techniques*.

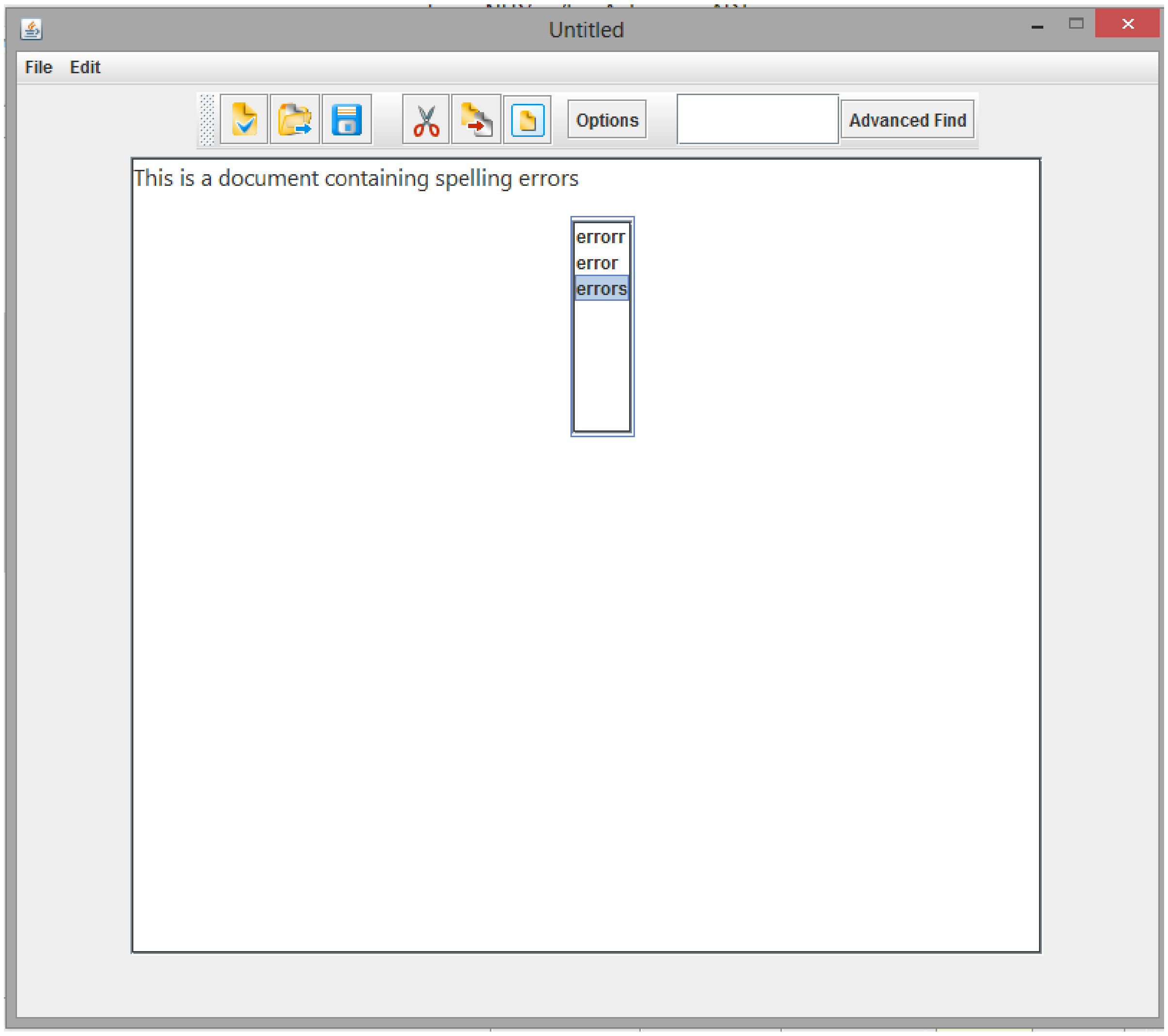
Integration testing by *white box technique* is facilitated through *System.output.println()* at appropriate positions in the java code to display corresponding outputs on ECLIPSE console. Absence or wrong output is indicative of coding errors. Such errors were identified and rectified.

Integration testing by *black box technique* has been done by changing the inputs for the integrated component and checking the corresponding outputs in order to ensure that the resulting integrated component works as expected in almost all conditions. Integration testing was done repeatedly to test for all possible conditions that might occur when deployed. All the possible mistakes at all corners were identified and rectified subsequently.

6.2.3 System testing

System testing is done on a completely integrated system in order to meet the expectations and requirements specified. The completely integrated system which in our case is the complete editor, was tested thoroughly to evaluate its compliance with specified requirements. The integrated component was run on different operating systems and its performance on each operating system was observed for possible different conditions. Based on these observations, the system was altered to enhance its running speed, thereby improving its performance as expected. Further, the altered system was checked to ensure its compliance.

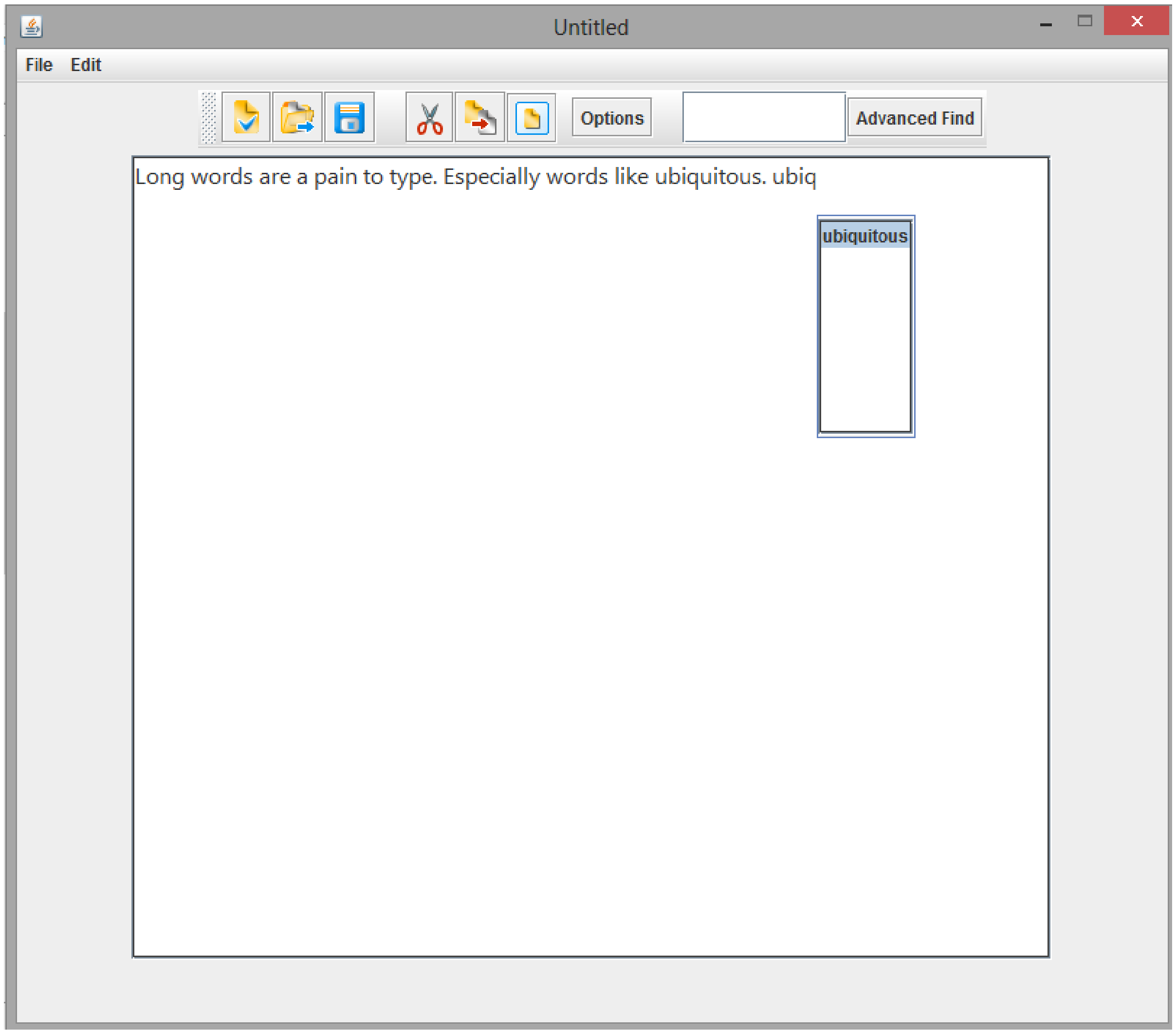
6.3 Snapshots



This is a document containing spelling errors

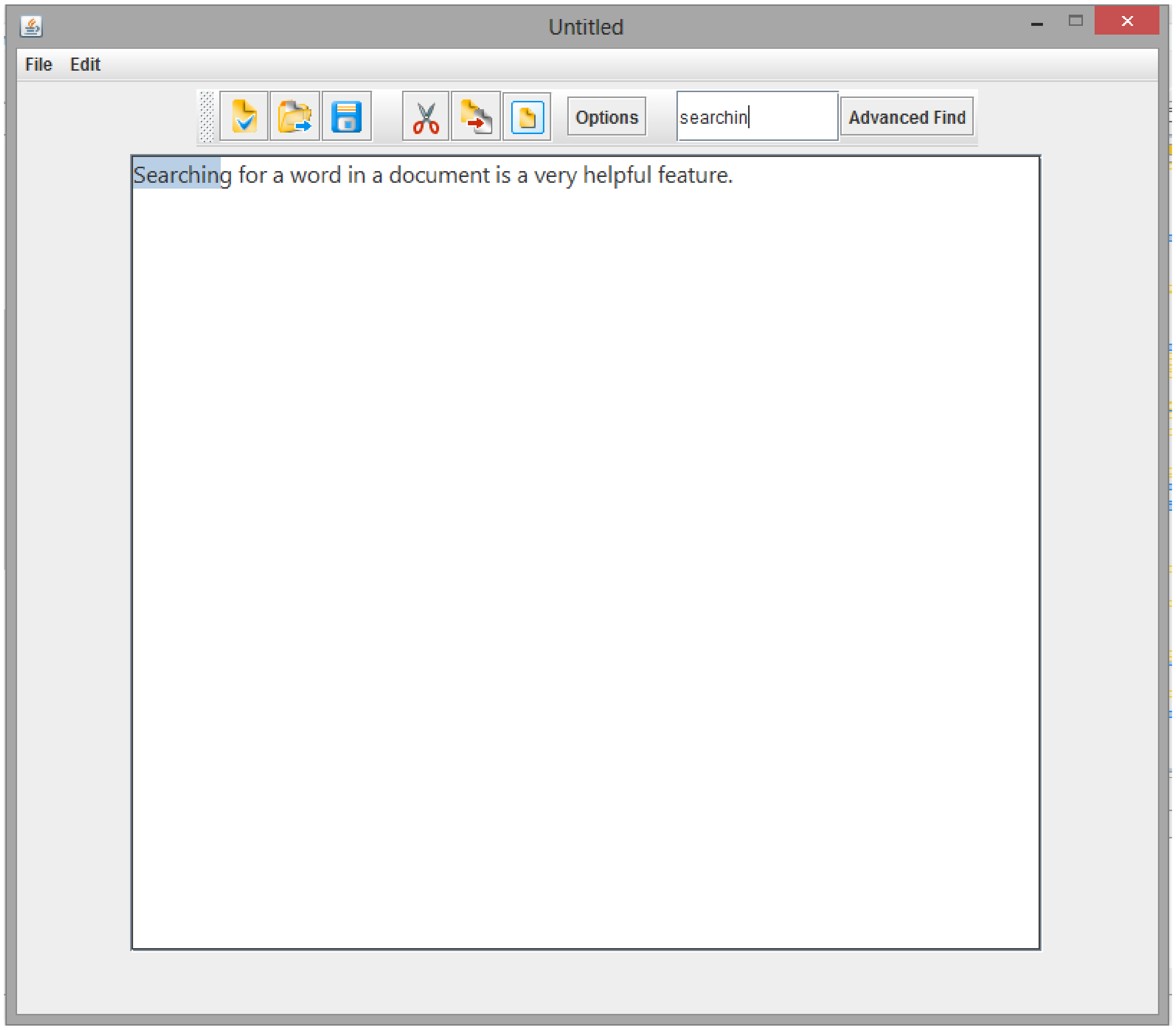
errorr
error
errors

Snapshot 1:
Spell Error detection and automatic correction
Candidate words shown for misspelt word, when a separator is pressed



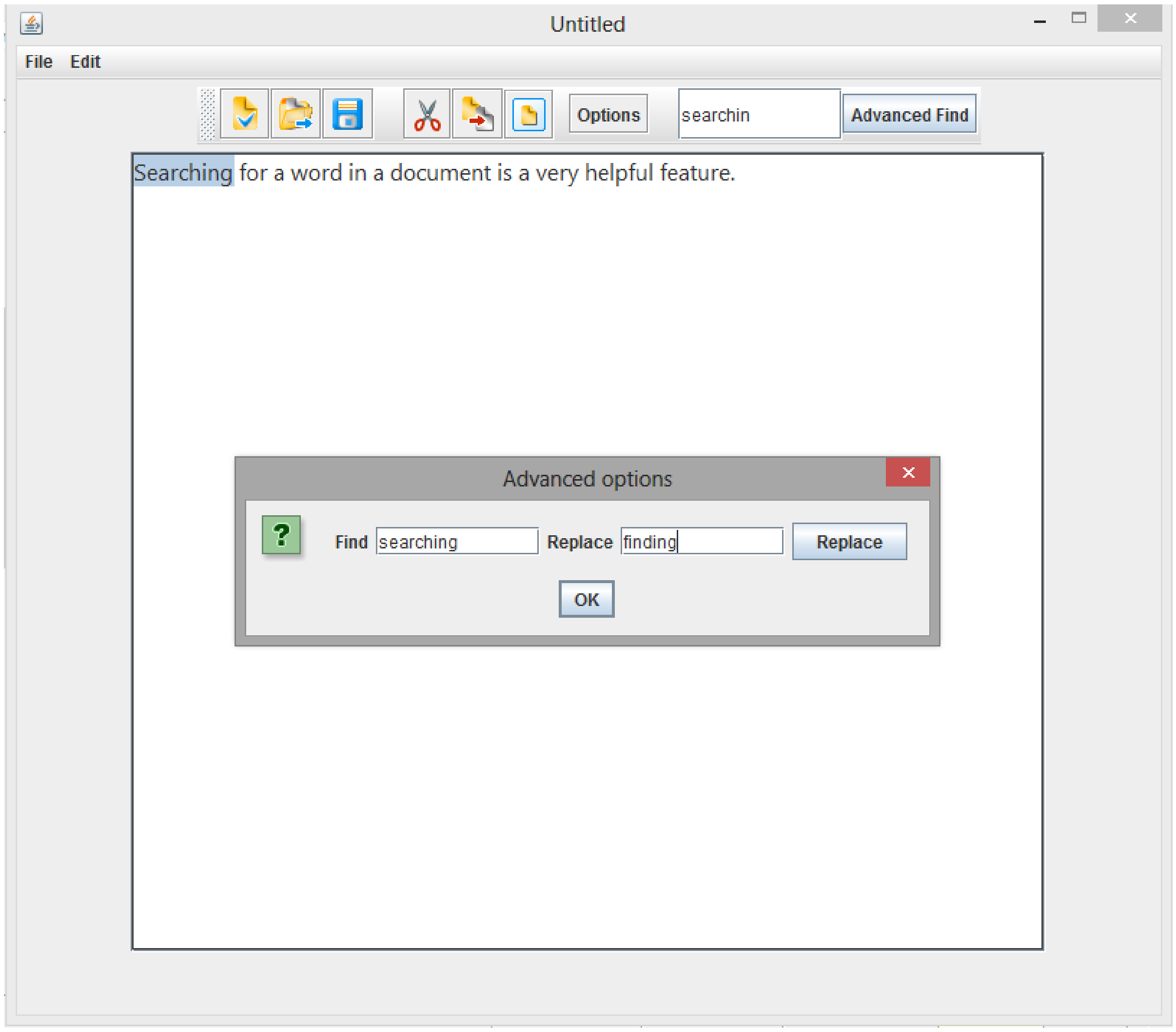
Long words are a pain to type. Especially words like ubiquitous. ubiq

Snapshot 2:
Auto complete
Completable words as a popup



Searching for a word in a document is a very helpful feature.

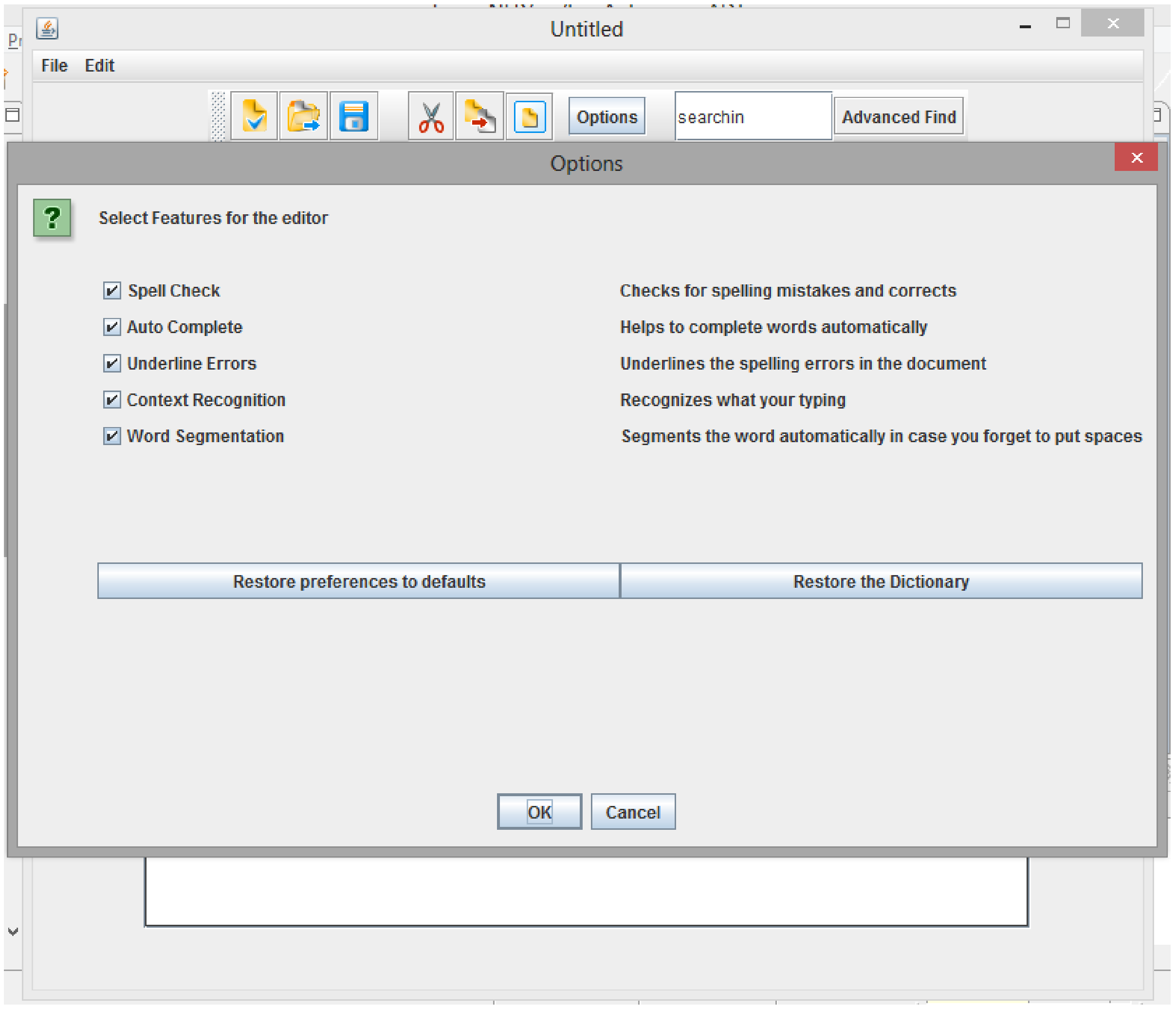
Snapshot 3:
Searching
Finding a word as you type



Searching for a word in a document is a very helpful feature.

Find: searching
Replace: finding

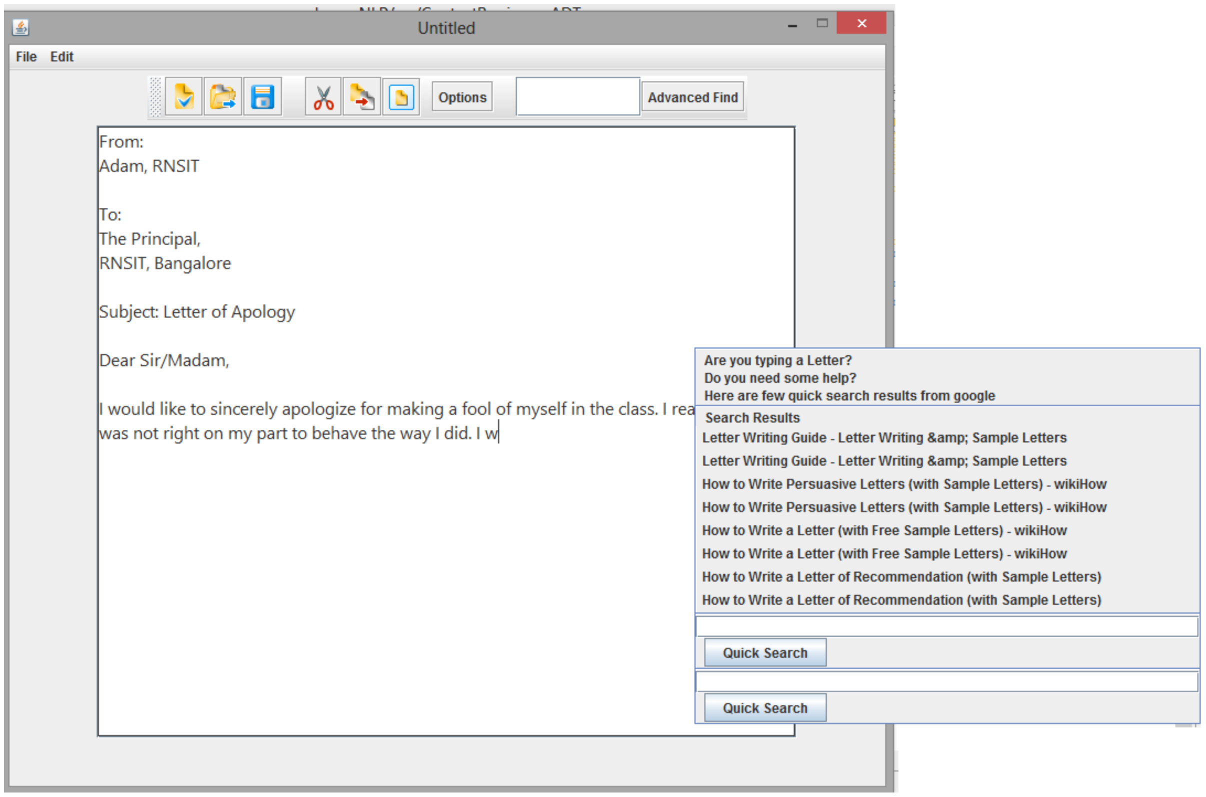
Snapshot 4:
Find and Replace
Option to find and replace.



Select Features for the editor

☑ Spell Check	Checks for spelling mistakes and corrects
☑ Auto Complete	Helps to complete words automatically
☑ Underline Errors	Underlines the spelling errors in the document
☑ Context Recognition	Recognizes what your typing
☑ Word Segmentation	Segments the word automatically in case you forget to put spaces

Snapshot 5:
Options
Ability to switch off certain features



From:
Adam, RNSIT

To:
The Principal,
RNSIT, Bangalore

Subject: Letter of Apology

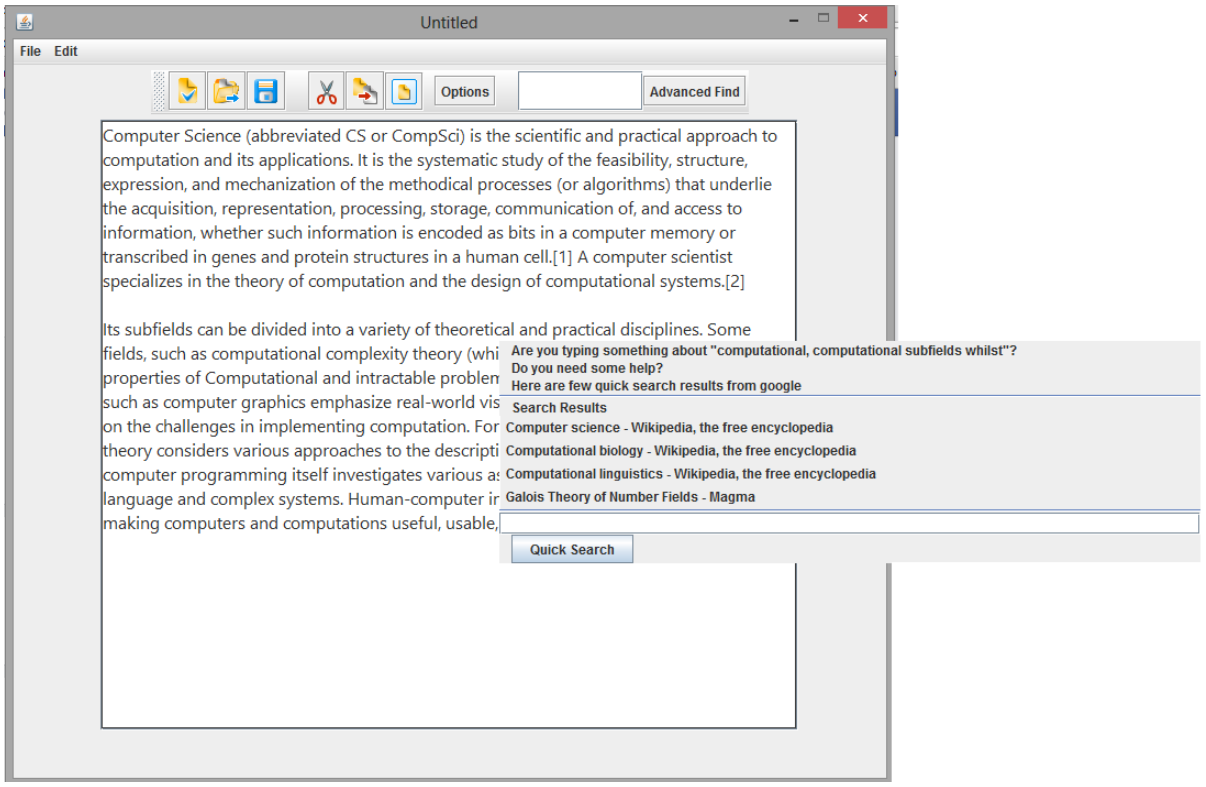
Dear Sir/Madam,

I would like to sincerely apologize for making a fool of myself in the class. I rea
was not right on my part to behave the way I did. I w

Are you typing a Letter?
Do you need some help?
Here are few quick search results from google

Search Results
Letter Writing Guide - Letter Writing &amp; Sample Letters
Letter Writing Guide - Letter Writing &amp; Sample Letters
How to Write Persuasive Letters (with Sample Letters) - wikiHow
How to Write Persuasive Letters (with Sample Letters) - wikiHow
How to Write a Letter (with Free Sample Letters) - wikiHow
How to Write a Letter (with Free Sample Letters) - wikiHow
How to Write a Letter of Recommendation (with Sample Letters)
How to Write a Letter of Recommendation (with Sample Letters)

Snapshot 6:
Letter Detection



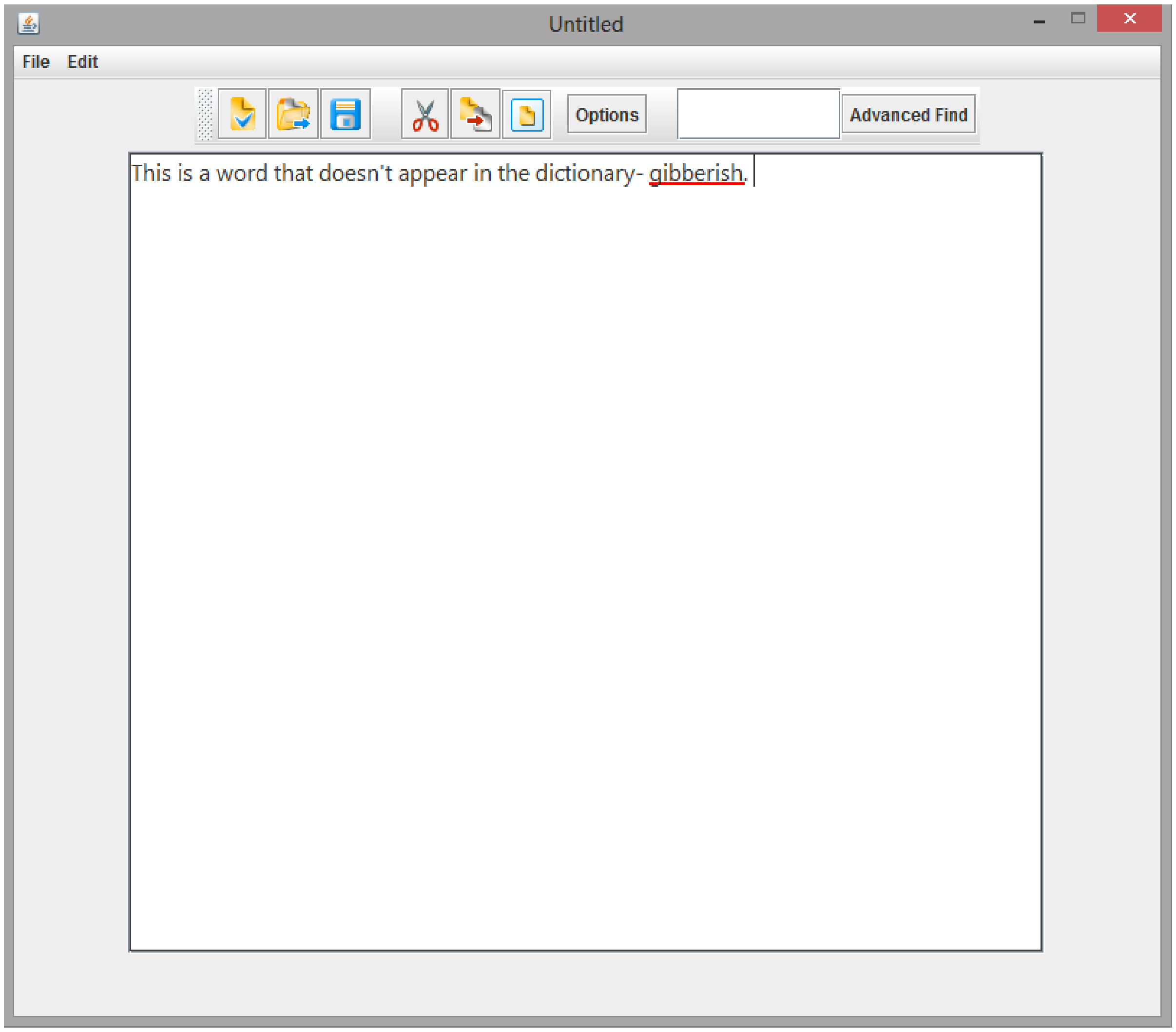
Computer Science (abbreviated CS or CompSci) is the scientific and practical approach to computation and its applications. It is the systematic study of the feasibility, structure, expression, and mechanization of the methodical processes (or algorithms) that underlie the acquisition, representation, processing, storage, communication of, and access to information, whether such information is encoded as bits in a computer memory or transcribed in genes and protein structures in a human cell.[1] A computer scientist specializes in the theory of computation and the design of computational systems.[2]

Its subfields can be divided into a variety of theoretical and practical disciplines. Some fields, such as computational complexity theory (whi
properties of Computational and intractable problem
such as computer graphics emphasize real-world vis
on the challenges in implementing computation. For
theory considers various approaches to the descripti
computer programming itself investigates various a
language and complex systems. Human-computer ir
making computers and computations useful, usable,

Are you typing something about "computational, computational subfields whilst"?
Do you need some help?
Here are few quick search results from google

Search Results
Computer science - Wikipedia, the free encyclopedia
Computational biology - Wikipedia, the free encyclopedia
Computational linguistics - Wikipedia, the free encyclopedia
Galois Theory of Number Fields - Magma

Snapshot 7:
Topic Extraction and quick search links provided as a popup



This is a word that doesn't appear in the dictionary- gibberish.

Snapshot 8:
Underlining misspelt word.

CHAPTER 7

CONCLUSION

The basic idea behind the project, was to do something new, something innovative, and something different. Natural Language Processing is an enormous subject with colossal scope. One of the basic applications of NLP is the concept of Text Processing, and we wanted to explore this sector as much as possible. MS Word is one of the finest examples of *"Total Text Processing"*. It is a complete package that takes care of almost all requirements that a user may have, while typing a document. But even MS Word has missed out on few features, and our aim was to exploit that.

After days of careful analysis, and needed brainstorming, we finalized on a set of features. Some of these feature are new and innovative, never attempted before (editor with AutoComplete, Regex find, Topic Extraction feature). And some are intriguing features that MS Word already owns, and we wished to emulate them, to satisfy our curiosity (spellchecking and correction). The aim was never to make a fancy good looking Editor, but to concentrate on, and implement powerful backend algorithms. Hence, the Editor was kept very simple, without extravagant design options.

The biggest challenge was to make this text editor, as *"Live"* as possible. Various algorithms were to react to every possible input from the user, and all the features available, had to be "*Live"* along with the editor.
The next challenge was to make this editor, as *"Fast and Smooth"* as possible. Using the best possible algorithms with least time complexities for the purpose, was the utmost priority. HashMaps and Entry Lists were some of the data structures used for the purpose. Smart ways of object creation, and using Serializable objects we used to save time. We also incorporated the usage of Background Processes to completely remove all possibilities of time consuming processes. Ultimately, "The Integrated Text Editor" is a radicle attempt to simplify the process of Text Editing.

BIBLIOGRAPHY

[1] Speech and Language Processing by Daniel Jurafsky, James H Martin

[2] A Spelling Correction Methodology Based on a Noisy Channel Model-Mark D. Kemighan,

[3] Kenneth W. Church, William A. Gale

[4] Natural Language Corpus Data: Beautiful Data by Peter Norvig

[5] Wikipedia

[6] Java: The complete reference, 8th edition

[7] Survey of Keyword Extraction Techniques- Brian Lott

[8] Stack Overflow

[9] The Brown Corpus

[10] Math ∩ Programming-A place for elegant solutions (www.jeremykun.com)