**CHAPTER 1**

**INTRODUCTION**

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](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/#b1) 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,[2](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/#b2)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[3](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/#b3) provided an estimate of the problem's difficulty, influencing the creation (1963) of Backus-Naur Form (BNF) notation.[4](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/#b4) BNF is used to specify a ‘context-free grammar’[5](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/#b5) (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*[6](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/#b6) used to specify text-search patterns. Regular expression syntax, defined by Kleene[7](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/#b7) (1956), was first supported by Ken Thompson's *grep* utility[8](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/#b8) on UNIX.

Subsequently (1970s), lexical-analyzer (lexer) generators and parser generators such as the *lex/yacc*combination[9](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/#b9) 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),[4](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/#b4) 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[11](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/#b11) 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[12](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/#b12)) 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 skeptical 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.

**CHAPTER 2**

**PROBLEM STATEMENT**

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

1. Spell Checker: A live automated spell checker that detects and corrects misspelt errors. 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 correct it.
2. Auto-complete: An option that continuously scans for long words and completes them automatically whenever applicable.
3. Dynamic find and replace: That provides an option to the user to search for a word or a character and replace it with another. Allows to search using regular expressions.
4. Context Recognition: To provide search links to the user to arrange for the privilege of easily searching for additional information on text being typed.

**CHAPTER 3**

**DESIGN AND IMPLEMENTATION**

**SPELL CHECK**

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.

**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

**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

**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 identified 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.

**Edit distance**

Edit distance is a way of quantifying 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:

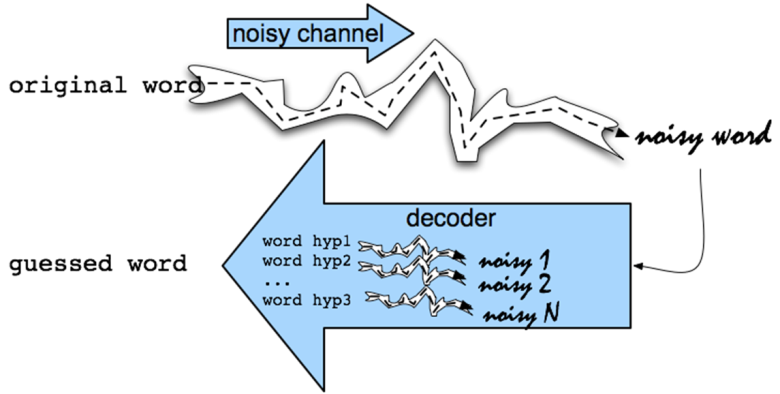
The Levenshtein distance between "kitten" and "sitting" is 3.

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.

**Noisy channel model for spelling correction**



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 produced given a misspelling w (this is done by producing a set of all words that are at an edit distance of 1 or 2 away from w). Our task is to select c ϵ C such that p(c|w) is maximium.

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

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

……………………………………………………………..Equation 2

We hence have to find c ϵ 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 ϵ C, higher is p(w|c). This means that if we find a c ϵ C that is an edit distance of 1 away from w, the corresponding p(w|c) is higher than that for those c ϵ 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.

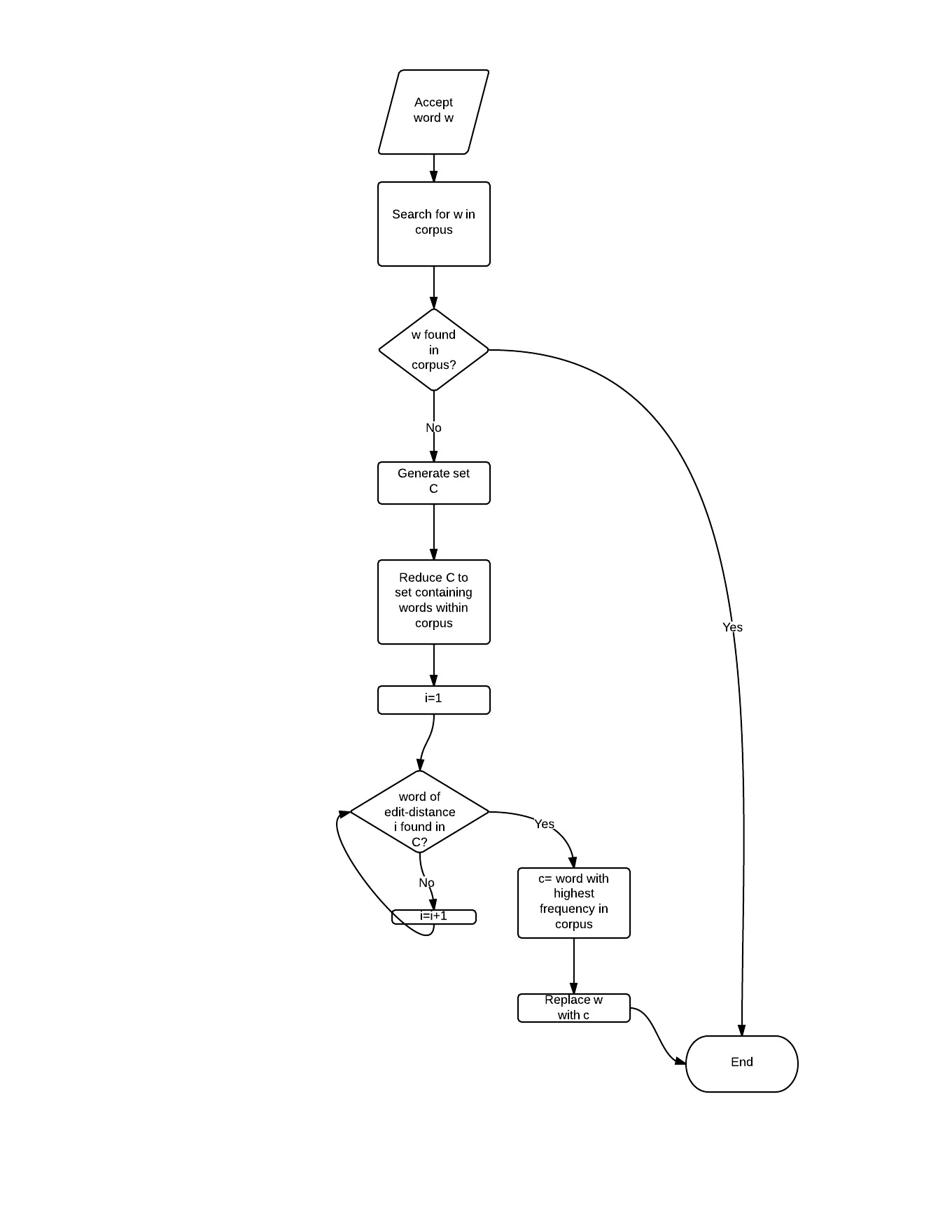
This intuitively means, given a non-word spelling error, we choose the word with highest similarity in terms of spelling to the misspelt word, provided it is fairly common in our corpus.

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 detect 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 ϵ C such that its edit distance from w is least and its corresponding frequency of occurrence in the corpus is highest.

**Flowchart**



**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.

**Methodology**

Given a word w of length c, the total number of segmentations possible is. 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 the value f= 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 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 is in performing word segmentation:

1. wordSequenceFitness(l)- A function that returns f for a list l comprising of . 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 maximum of those segmentations 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.

**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 ofthe 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.

**Autocomplete**

The text editor provides a handy feature called autocomplete. That helps the user type faster by remembering ling words that have already been typed. Often when writing a document about a particular topic, certain terms or names are repeated. 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.

**Working**

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. 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. This process is live and hence does not hamper the typist’s typing routine.

**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 concerning it 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.

**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 hot 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.

**Essay topic detection using keyword extraction or Header extraction**

If the document is found to not fall under any of the three document types (letter, resume, 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.

**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*= where is the frequency of term t in the current document d and 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.

Links are provided to the topics/keywords using Java’s Gson Module.

**DYNAMIC 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.**

**Regular expression**

In theoretical computer scienceand 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.

|  |  |
| --- | --- |
| METACHARACTER | 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 |

**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, for example: "[a]" and "[b]".
* s.\* matches any number of characters preceded by s, for example: "saw" and "seed".

**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 ):** Used to create instance of the pattern.

1. **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.DefaultPainter);

} 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).

**REPLACE**

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

**Implementation**

Implemented using default packages of java and its utilities in a **Brute force approach.**

**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.

**To find position of a word or character**

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 ):** Used to create instance of the pattern.

1. **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**

**/\***

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.

**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.

**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.

**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.

**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 Acessible**

**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.

**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 source-compatible 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.**

**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.

**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.