Context-Sensitive Spelling Correction

A Project Report
Submitted in the partial fulfillment of the requirements for the award of the degree of

Bachelor of Technology in Computer Engineering

Under the supervision of:
Dr. M. N. Doja
Professor
Department of Computer Engineering
Jamia Millia Islamia

Submitted by:
Ashhar Hasan (13BCS-0015)
Adeela Izhar (13BCS-0006)

Department of Computer Engineering
Faculty of Engineering and Technology
Jamia Millia Islamia, New Delhi — 110025

Contents

List of Figures

List of Tables

Certificate

This is to certify that the project report entitled "Context-Sensitive Spelling Correction", is an authentic work carried out by **Ashhar Hasan** and **Adeela Izhar**.

The work is submitted in partial fulfillment of the requirements for the award of the degree of *Bachelor of Technology in Computer Engineering* under my guidance. The matter embodied in this project work has not been submitted earlier for the award of any degree or diploma to the best of my knowledge and belief.

19/12/2016

Dr. M. N. Doja
Professor
Department of Computer Engineering
Jamia Millia Islamia

Acknowledgement

We would like to thank our mentor **Dr. M. N. Doja** (*Professor, Department of Computer Engineering*) for giving us the opportunity to undertake the project and bring it to completion. We thank him for his guidance, insights and advices and appreciate his engagement.

We would also like to thank **Dr. Tanvir Ahmad** (*Head of Department, Department of Computer Engineering*) for allowing us to undertake the project of our choice and assigning a great mentor for the same.

We would like to extend special gratitude to the assistants and lab coordinators of the Department of Computer Engineering for providing us the infrastructure necessary to work efficiently on the project.

Ashhar Hasan (13BCS-0015)
Adeela Izhar (13BCS-0006)

Abstract

Context-Sensitive Spelling Correction is the task of fixing spelling errors that result in valid words, such as `I'd like to eat desert`, where `desert` was typed when `dessert` was intended. These errors will go undetected by conventional spell checkers, which only flag words that are not found in a predetermined word list or a dictionary.

Context-Sensitive Spelling Correction involves learning to characterize the linguistic contexts in which different words such as `dessert` and `desert`, tend to occur. The problem is that there is a multitude of features one might use to characterize these contexts: features that test for the presence of a particular word nearby the target word; features that test the parts of speech around the target word; and so on. In general, the number of features will range from a few hundred to over ten thousands. A good method to filter out features is needed because even after filtering or removing noisy features, we will still end up with a large feature space, while the target concept (a context in which `desert` can occur in this case) depends only on a small subset. For this problem, the class of multiplicative weight algorithms like *Winnow Algorithm* have been shown to have exceptionally good theoretical properties.

In the work reported here, we present an algorithm that combines variants of Winnow and applies it to the problem of *Context-Sensitive Spelling Correction*. We use the *Winnow Algorithm* to find the best set of features which will then be used to train linear classifiers which try to correct such contextual mismatches using the learnt features.

Introduction

Writing is one of the predominant and vital ways of communication through which humans can express their views to others and keep a record. It is one of the most effective forms of language representation. It does the task of representing language by engraving signs and symbols. Writing is composed of vocabulary, semantics and grammar. Text is considered as the outcome of writing.

The scalability of writing text has increased due to the availability of computers, due to which many issues such as spelling mistakes have also evolved. Mistakes can sidetrack readers from the efforts the writer has put in his writing. Therefore it becomes important to remove these mistakes. Hence, it prompted the need to use spelling checkers so that errors can be minimized while writing. Spell checkers are either part of large applications, for instance, search engines, email clients etc. or a standalone application that is efficient in performing correction on a piece of text. Nearly all word processors have a built-in spelling checker that flags the spelling mistakes. It also provides the solution to correct these spelling mistakes by choosing a possible alternative from a given list. For identification of spelling mistakes, most spell checkers check each word drawn separately from the written text against the dictionary-stored words. If the word is found while searching the dictionary, it is considered as correct word regardless of its context. This approach is efficient for identifying the non-word spelling mistakes but other mistakes cannot be identified using this method. The other mistakes are real-word spelling mistakes.

Real-word spelling mistakes are words that are correctly spelled but are not intended by the user. Mistakes under this category go unrecognized by most spell checkers because they handle non-word spelling mistakes by checking against the dictionary word list only. This technique is effective to identify the non-word spelling mistakes but not the real-word spelling mistakes. To identify the real-word spelling mistakes, there is a need to utilize the neighboring contextual information of the target word. An ex of such a sentence is `“I want to eat a peace of cake”` and the confused word set in this case is `{peace, piece}`. To identify that `‘peace’` cannot be used in this case, we utilize the neighboring contextual information `‘cake’` for word `‘piece’`.

Problem Statement and Approach

2.1 Context-Sensitive Spelling Correction

With the widespread availability of spell checkers to fix errors that result in non-words, such as `teh`, the predominant type of spelling error has become the kind that results in a real, but unintended word; for example, typing `there` when `their` was intended. Fixing this kind of error requires a completely different technology from that used in conventional spell checkers: it requires analyzing the context to infer when some other word was more likely to have been intended. We call this the task of *Context-Sensitive Spelling Correction*. The task includes fixing not only "classic" types of spelling mistakes, such as *homophone errors* (e.g. `peace` and `piece`) and *typographic errors* (e.g. `form` and `from`), but also mistakes that are more commonly regarded as *grammatical errors* (e.g. `among` and `between`), and errors that cross *word boundaries* (e.g. `maybe` and `may be`).

The problem started receiving attention in the literature within about the last dozen years. A number of methods have been proposed, either for *Context-Sensitive Spelling Correction* directly, or for related lexical disambiguation tasks.

The methods include:

Word Trigrams [**?**],
Bayesian Classifiers [**?**],
Bayesian Hybrids [**?**],
A combination of Part-of-Speech Trigram and Bayesian Hybrids [**?**],
Transformation-Based Learning [**?**],
Latent Semantic Analysis [**?**],
Decision Lists [**?**],
Noisy Channel Algorithms [**?**],
Language Models [**?**],
OCR to imrpove training [**?**].

We believe that a *Winnow Algorithm* based approach is particularly promising for this problem, due to the problem's need for a very large number of features to characterize the context in which a word occurs, and *Winnow Algorithm's* theoretically-demonstrated ability to handle such large numbers of features.

2.2 Problem Formulation

We treat *Context-Sensitive Spelling Correction* as a task of word disambiguation. The ambiguity among words is modelled by *confusion sets*. A confusion set $C = \{w_1, ..., w_n\}$ means that each word w_i in the set is ambiguous with each other word w_j in the set C. Thus if $C = \{desert, dessert\}$, then when the spelling-correction program ses an occurrence of either $desert$ or $dessert$ in the target document, it takes it to be ambiguous between $desert$ and $dessert$, and tries to infer from the context which of the two it should be.

This treatment requires a collection of confusion sets to start with. There are several ways to obtain such a collection. One is based on finding words in the dictionary that are one typo away from each other [**?**]. Another finds words that have the same or similar pronunciations. Since this was not the focus of the work we did, we simply took (most of) our confusion sets from the list of "Common Errors in English Usage" from the list curated by Paul Brians [**?**].

A final point concerns the two types of errors a spelling-correction program can make: false negatives (complaining about a correct word), and false positives (failing to notice an error). We will make the simplifying assumption that both kind of errors are equally bad. In practice, however, false negatives are much worse, as users get irritated by the programs that badger them with bogus complaints. However, given the probabilistic nature of the methods that will be presented below, it would not be hard to modify them to take this into account. We would merely set a confidence threshold, and report a suggested correction only if the probability of the suggested word exceeds the probability of the user's original spelling by at least a threshold amount. The reason this was not done in the work reported here is that setting the confidence threshold involves a certain subjective factor (which depends on the user's "irritability threshold"). Our simplifying assumption allows us to measure performance objectively, by the single parameter of prediction accuracy.

2.3 Problem Representation

2.3.1 Feature Selection

A target problem in *Context-Sensitive Spelling Correction* consists of:

1. A sentence
2. A target word in that sentence to correct

Both the *Bayesian* and *Winnow-based* algorithms studied here represent the problem as a list of active features. Each active feature indicates the presence of a particular linguistic pattern in the context of the target word.

We use two types of features:

Context words: *Context-word* features test for the presence of a particular word within $\pm k$ words of the target word.

Collocations: *Collocations* test for a pattern of up to l contiguous words and/or part-of-speech tags around the target word.

In the experiments reported here, k was set to 10 and l was set to 2. Examples of useful features for the confusion set $\{weather, whether\}$ include:

1. $cloudy$ within ± 10 words
2. $\underline{\quad\quad}\ to\, VERB$

Feature (1) is a context-word feature that tends to imply $weather$. Feature 2 is a collocation that checks for the pattern *"to VERB"* immediately after the target word, and tends to imply $whether$ as in (I don't know whether to laugh to cry).

2.3.1.1 Context Words

One clue about the identity of an ambiguous target word comes from the words around it. For instance, if the target word is ambiguous between $desert$ and $dessert$, and we see words like $arid$, $sand$ and sun nearby, this suggests that the target word should be $desert$. On the other hand, words such as $chocolate$ and $delicious$ in the the context imply $dessert$. This observation is the basis for the method of context words. The idea is that each word w_i in the confusion set will have a characteristic distributions of words that occur in its context; thus to classify an ambiguous target word, we look at the set of words around it and which w_i's distribution they most closely follow. The main parameter to tune for the method of context words is k, the half-width of the context windows. Previous work [**?**] shows that smaller values of k (3 or 4) work well for resolving local syntactic ambiguities, while larger values (20 to 50) are suitable for resolving semantic ambiguities. We tried the values 3, 6, 9, 10, 12, 16, 18, 20 and 24 on our confusion sets and found that $k = 10$ worked the best in general cases. In the rest of the document, this value of k will be used when referring to k.

$k = 10$	But, $\underline{\text{how}}$ I sometimes need a moment of rest, and *peace*
$k = 8$	No matter $\underline{\text{how}}$ earnest is out quest for guaranteed *peace*
$k = 6$	$\underline{\text{How}}$ best to destroy your *peace*?

Figure 2.1: Effect of different values of k on context words

Training Phase

1. Propose all words as candidate context words.
2. Count occurrences of each candidate context word in the training corpus.
3. Prune context words that have insufficient data or are uninformative.
4. Store the remaining context words and their associated statistics for use at run time.

Run Time

1. Initialize the probability for each word in the confusion set to its prior probability.
2. Go through the list of context words that was saved during training. For each context word that appears in the context of the ambiguous target word, update the probabilities.
3. Choose the word in the confusion set with the highest probability.

Figure 2.2: Outline of the method of using *context words*

2.3.1.2 Collocations

The method of context words is good at capturing generalities that depend on the presence of nearby words, but not their order. When order matters, other more syntax-based methods, such as collocations and trigrams, are appropriate. We used collocations to capture order dependencies. A collocation expresses a pattern of syntactic elements around the target word. We allow two types of syntactic elements: words and part-of-speech tags.
Going back to the $\{desert, dessert\}$ example, a collocation would imply $desert$ might be:

$$PREP\ the\ \underline{\qquad}$$

This collocation would match the sentences:

Travellers entering from the *desert* were confounded…
…along with some guerilla fighting in the *desert*.
…two ladies who lay by pinkly beside him in the *desert*…

Matching part-of-speech tags against the sentence is done by first tagging each word in the sentence with its set of possible part-of-speech tags obtained from a dictionary and a language model. For instance, $walk$ has the tag set $\{NS, V\}$ corresponding to its use as a singular noun and as a verb. For a tag to match a word, the tag must be a member of the word's tag set. The reason we use tag sets instead of running a tagger on the sentence to produce unique tags is that taggers need to look at all the words in the sentence which doesn't make a lot of sense when the target word in the sentence itself is ambiguous.

The method of collocations was implemented in much the same way as the method of context words. The idea is to discriminate among the words w_i in the confusion set by identifying the collocations that tend to occur around each w_i. An ambiguous target word is then classified by finding all collocations that match its context. Each collocation provides some degree of evidence for each word in the confusion set.

Like the method of context words, the method of collocations has one main parameter to tune: l, the maximum number of syntactic elements in a collocation. Since the collocations grow exponentially with l, we varied l from 1 to 3 and finally settled on 2.

2.3.2 Intuition for Feature Selection

The intuition for using these two types of features is that they capture two important, but complementary aspects of context. Context words tell us what kind of words tend to appear in the vicinity of the target word — the "lexical atmosphere". They therefore capture aspects of the context with a wide-scope, semantic flavour, such as discourse topic and tense. Collocations, in contrast, capture the local syntax around the target word. Similar combinations of features have been used in related tasks, such as accent restoration [**?**] and word-sense disambiguation [**?**].

2.3.3 Feature Extraction

We use a *feature extractor* to convert from the initial text representation of a sentence to a list of active features called a *feature vector*. The feature extractor has a preprocessing phase in which learns a set of features for the task. Thereafter, it can convert a sentence into a list of active features simply by matching its set of learned features against the sentence.

In the preprocessing phase, the feature extractor learns a set of features that characterize the contexts in which each word w_i in the confusion set tends to occur. This involves going through the training corpus, and, each time a word in the confusion set occurs, generating all possible features for the context — namely, one context-word feature for every distinct word within $\pm k$ words, and one collocation for every way of expressing a pattern of up to l contiguous elements. This gives an exhaustive list of all features found in the training set.

2.3.4 Feature Pruning

Statistics of occurrence of the features are collected in the process as well. At this point, pruning criteria may be applied to eliminate unreliable or uninformative features.
We use two criteria (which make use of the aforementioned statistics of occurrence):

1. The feature occurred in practically none or all of the training instances (specifically, it had fewer than 10 occurrences or fewer than 10 non-occurrences)
2. The presence of the feature is not significantly correlated with the identity of the target word (determined by a *chi-square test* at the 0.05 significance level).

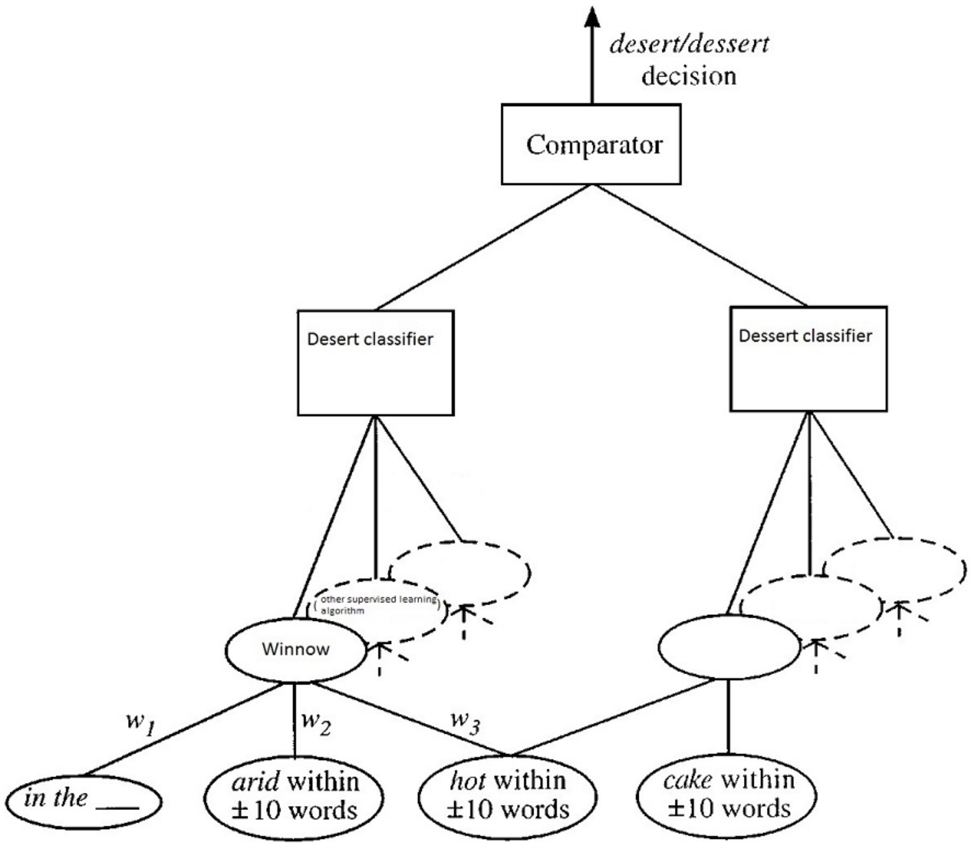


Figure 2.3: Prediction of a sentence using Winnow

Terminology and Algorithm

3.1 Winnow Algorithm

3.1.1 Basic Idea

Put simply, *Winnow* is a machine learning technique for discerning the disjunction, (i.e. the **OR**ing function) from labelled examples. Concretely, what this means is, given m labelled samples, each with a vector of n Boolean features $x = \{x_1, \ldots, x_n\}$, Winnow learns which features to union to get the correct label.

Take the example of email spam filtering, on the basic level, we will take the body of the many emails, and tokenize them into array of words, this will be our features vector.

Each feature (word) is denoted by 1 if it exists in the example, 0 if it doesn't.

x_1	x_2	x_3	x_4	x_5	$\ldots$	x_n	spam (yes/no)
\$\$\$	100%	$free$	$viagra$	the	$weight\,loss$	$\ldots$	$f(x)$
0	1	0	1	0	1	$\ldots$	$?$

Now we can see the problem, the feature set can be very large, with many features set to $true$ (the for example occurs in almost every email) yet the correct labelling depends only on a small subset of features r.

3.1.2 Working

1. Initialize the weights $w_1 = w_2 = \ldots = w_n = 1$ on the n variables (features vector).
2. Given an example $x = \{x_1, \ldots, x_n\}$, output 1 if:
   $$\sum_{i=1}^{n} w_i\, x_i \geq \theta$$
   else output 0.
   θ is a threshold value we set to our discretion.
3. If the algorithm makes a mistake:
   (a) If it predicts 0 when $f(x) = 1$, then for each $x_i = 1$, increase the value of w_i. (value was too low, promote contributing features)

(b) If it predicts 1 when $f(x) = 0$, then for each $x_i = 1$, decrease the value of w_i. (value was too high, demote contributing features)

Winnow gives us a mistake bound of $O(r\, log(n))$. Where, r is the number of relevant features and n is the total number of features.

3.2 Data Serialization

In computer science, in the context of data storage, serialization is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection) and reconstructed later in the same or another computer environment. When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object. For many complex objects, such as those that make extensive use of references, this process is not straightforward. Serialization of object-oriented objects does not include any of their associated methods with which they were previously inextricably linked. This process of serializing an object is also called *marshalling* an object. The opposite operation, extracting a data structure from a series of bytes, is deserialization (which is also called *unmarshalling*).

Data serialization allows to perform operations on test and training data "in-memory" instead of reading them from disk, thus increasing speed. Serializing the data structure in an architecture independent format means preventing the problems of byte ordering, memory layout, or simply different ways of representing data structures in different programming languages. Inherent to any serialization scheme is that, because the encoding of the data is by definition serial, extracting one part of the serialized data structure requires that the entire object be read from start to end, and reconstructed. In applications where higher performance is an issue, it can make sense to expend more effort to deal with a more complex, non-linear storage organization. This algorithm uses an XML representation for the objects. Several object-oriented programming languages directly support object serialization (or object archival), either by syntactic sugar elements or providing a standard interface for doing so.

```
<?xml version=”1.0”?>
<ArrayOfSentence
 xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
 xmlns:xsd=”http://www.w3.org/2001/XMLSchema”>
    <Sentence>
        <Words>
            <string>0%</string>
            <string>agarose</string>
            <string>gel</string>
        </Words>
        <PosTags>
            <string>JJ</string>
            <string>NN</string>
            <string>NN</string>
        </PosTags>
    </Sentence>
</ArrayOfSentence>
```

```
<ArrayOfComparator>
  <Comparator>
    <Cloud>
      <d3p1:KeyValueOfstringArrayOfanyTypety7Ep6D1>
        <d3p1:Key>council</d3p1:Key>
        <d3p1:Value>
          <d3p1:anyType>
            <d6p1:Demotion>0.5</d6p1:Demotion>
            <d6p1:Mistakes>457</d6p1:Mistakes>
            <d6p1:Promotion>1.5</d6p1:Promotion>
            <d6p1:Threshold>1</d6p1:Threshold>
            <d6p1:Weights>
              <d3p1:double>2.4892E-60</d3p1:double>
              <d3p1:double>6.5253E-55</d3p1:double>
              <d3p1:double>8.0779E-28</d3p1:double>
            </d6p1:Weights>
          </d3p1:anyType>
        </d3p1:Value>
      </d3p1:KeyValueOfstringArrayOfanyTypety7Ep6D1>
    </Cloud>
  </Comparator>
</ArrayOfComparator>
```

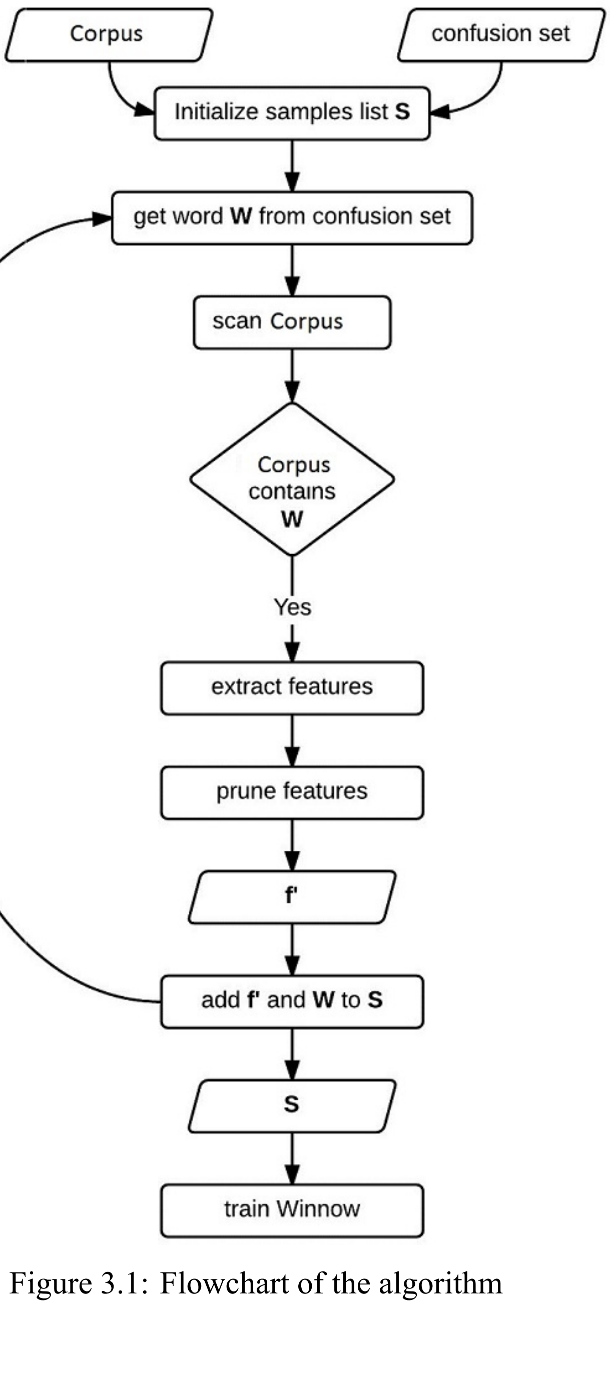


Figure 3.1: Flowchart of the algorithm

Methodology

4.1 Data Acquisition

In the experiments that follow, the training and test sets were drawn from the written part of the American National Corpus (OANC) which is about $14\ million$ words in total, around $5\ million$ sentences. We split the data into a $80 - 20$ split by sentences and used 28 confusion sets taken from *Common Errors in English Usage* [?] having appreciable frequency in our corpus.

The data was further divided into a *Debug Corpus*, a *Release Corpus* and a *Test Corpus*. The *Release Corpus* is reduced version of the original corpus (*Debug Corpus*) to be used to easily evaluate effect of changes made during development without needing to spend time on retraining the entire model. The *Test Corpus* is the test data used to find accuracy of the classifier. Randomized sets were also drawn during testing and training to make sure that the model did not overfit.

Confusion Sets	**Word**	**Count**
plaine-plane	plaine	84
	plane	517
threw-through	threw	923
	through	7210
by-buy-bye	by	60361
	buy	12440
	bye	11439
vain-vane-vein	vain	92
	vane	32
	vein	219
cite-site-sight	cite	731
	site	4287
	sight	917
loose-lose	loose	260
	lose	550
peace-piece	peace	1571
	piece	2577
weak-week	weak	810
	week	3439
coarse-course	coarse	250
	course	3168

knew-new	knew new	3695 18233
brake-break	brake break	82 615
hour-our	hour our	3245 11588
to-too-two	to too two	201119 29787 39437
weather-whether	weather whether	1010 4583
where-were	where were	14568 41712
hear-here	hear here	1430 7135
fourth-forth	fourth forth	617 443
their-there	their there	23008 19501
complement-compliment	complement compliment	141 50
sea-see	sea see	1961 9343
desert-dessert	desert dessert	261 81
council-counsel	council counsel	682 731
later-latter	later latter	2761 1051
waist-waste	waist waste	106 293
passed-past	passed past	959 2586
principal-principle	principal principle	543 684
quiet-quit-quite	quiet quit quite	567 462 1621
rain-reign-rein	rain reign rein	309 138 64

4.2 Data Preprocessing

4.2.1 Non-Word Error Correction

The first sub-step is correction of non-word spelling errors (perhaps by a conventional spelling checker). This should occur before context errors are sought (rather than after or in parallel), in order to maximize the number of words in the text available to check for semantic relatedness to each word considered by the algorithm. Moreover, as we observed earlier, it is not unusual for context errors to be introduced during conventional spelling checking, so context error detection should follow that.

For this stage we used an *Edit Distance* based algorithm to correct non-word errors. We used the *GNU Aspell* dictionary for our purposes.

4.2.1.1 Edit Distance Algorithm

The Levenshtein Distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other.

Mathematically, the Levenshtein Distance between two strings a and b is given by:

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{a_i \neq b_j} \end{cases} & \text{otherwise} \end{cases} \tag{4.1}$$

Where $1_{(a_i \neq b_j)}$ is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise, and $lev_{a,b}(i,j)$ is the distance between the first i characters of a and the first j characters of b.

For example, the Levenshtein distance between "*kitten*" and "*sitting*" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

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).

4.2.2 Identification of Context Errors

The second sub-step identifies words in the document that are to be checked for error by removing from further consideration those that are not in the lexicon at all and those that are on a stop-list. The algorithm, by its nature, applies only to words whose meaning or meanings are known and have content that is likely to be topical. We therefore exclude closed-class words and common non-topical words. Closed-class words are excluded as their role in a text is almost always purely functional and unrelated to content or topic. It is of course possible that a typing error could turn a highly contentful word into a closed-class word or vice versa; but the former case will not be considered by the algorithm and the latter will be considered but not detected. The exclusion of 'untopical' open-class words, such as $know$, $find$, and $world$, is well-precedented in information retrieval. Here, there is a trade-off between making the list as short as possible, in order to let as many words as possible be checked, and making the list as long as possible in order to avoid spurious relationships.

4.3 Feature Extraction Algorithm

We use a *feature extractor* to convert from the initial text representation of a sentence to a list of active features called a *feature vector*. The feature extractor has a preprocessing phase in which learns a set of features for the task. Thereafter, it can convert a sentence into a list of active features simply by matching its set of learned features against the sentence.

In the preprocessing phase, the feature extractor learns a set of features that characterize the contexts in which each word w_i in the confusion set tends to occur. This involves going through the training corpus, and, each time a word in the confusion set occurs, generating all possible features for the context — namely, one context-word feature for every distinct word within $\pm k$ words, and one collocation for every way of expressing a pattern of up to l contiguous elements. This gives an exhaustive list of all features found in the training set.

4.3.1 Corpus Preprocessing

Corpus is first split into complete sentences; each sentence is tokenized into list of words and pos tags. For this purpose, Apache's OpenNLP library's .NET framework port was used.

```
public class Sentence
{
    public string[] Words { get; set; }
    public string[] POSTags { get; set; }
}
```

```
private IEnumerable<Sentence> PreProcessCorpora(
    IEnumerable<string> corpora)
{
    var sentences = new ConcurrentBag<sentence>();
    Parallel.ForEach(corpora, phrase =>
    {
        string[] tokens = SplitIntoWords(phrase), posTags;
        lock (_lock)
        { posTags = _posTagger.Tag(tokens); }
        sentences.Add(new Sentence
        { Words = tokens, POSTags = posTags });
    });

    return sentences;
}
```

4.3.2 Getting Context and Collocation Features

Combing through the training corpus (i.e. for each sentence), each time a word in the confusion set occurs, get all possible features for the context—namely, one context-word feature for every distinct word within $\pm k$ words and 2 collocations (occurring before and after word) for every way of expressing a pattern of up to l contiguous elements.

Statistics of occurrence of the features are collected in the process as well, namely, for each feature F we collect for target word W, we record:

N11 Number of documents (sentences) containing both F and W.

N10 Number of documents containing F only.

N01 Number of documents containing W only.

N00 Number of documents where neither W nor F were found.

N Total number of documents.

4.3.3 Feature Pruning Algorithm

After these steps, an exhaustive list of all features found in the training corpus is generated.At this point, pruning criteria may be applied to eliminate unreliable or uninformative features.
We use two criteria (which make use of the aforementioned statistics of occurrence):

1. The feature occurred in practically none or all of the training instances (specifically, it had fewer than 10 occurrences or fewer than 10 non-occurrences)
2. The presence of the feature is not significantly correlated with the identity of the target word (determined by a *chi-square test* at the 0.05 significance level).

4.4 Training and Testing

4.4.1 Training

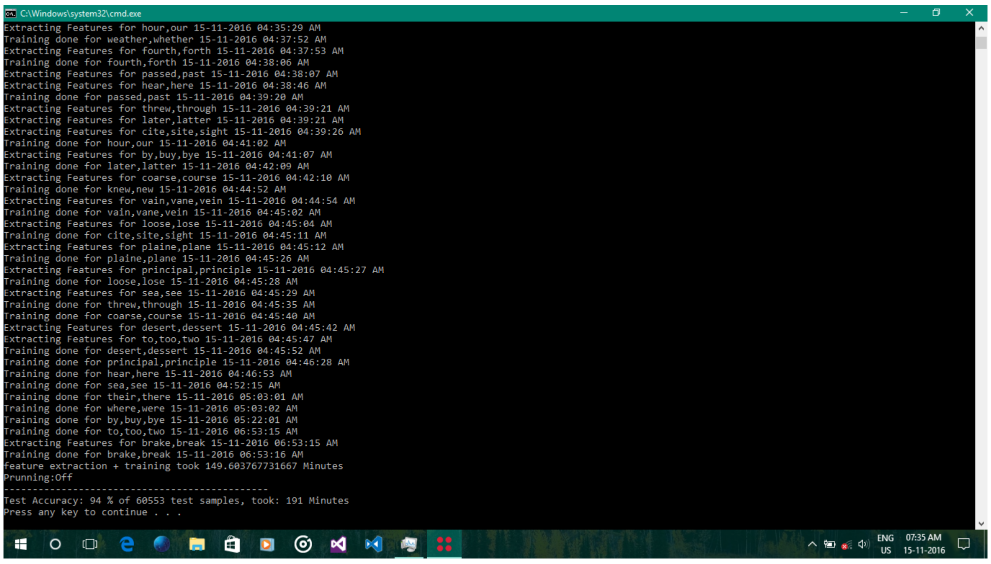
1. Propose all possible features as candidate features.
2. Prune features that have insufficient data or are uninformative discriminators.
3. Sort remaining features in order of decreasing strength.
4. Store the list of features and their associated statistics for use at run time.
5. Serialize and write the trained model and feature vectors to a file that can be distributed.

A training example consists of a sentence, represented as a set of active features, together with the word Wc in the confusion set that is correct for that sentence.

Now that we have a list of proper training data, we train a classifier for each word in our confusion set. Training proceeds in an on-line manner, each example is treated as a positive example for the classifiers for W_c, and as a negative example for the classifiers for the other words in the confusion set.

```
private class RoughSample
{
    public HashSet<string> Features { get; set; }
    public string Word { get; set; }
}

var cloudClassifiers = new Dictionary<string,
   ISupervisedLearning[]>(confusionSet.Count());
foreach (var word in confusionSet)
{
    var classifiers = new ISupervisedLearning[1];
    int i = 0;
    for (double beta = 0.5; beta < 1 && i < classifiers.Length;
         beta += 0.1, i++)
    { classifiers[i] = new Winnow.Winnow(features.Length,
        threshold: 1, promotion: 1.5, demotion: beta,
        initialWeight: 1); }
    cloudClassifiers[word] = classifiers;
|| }
```



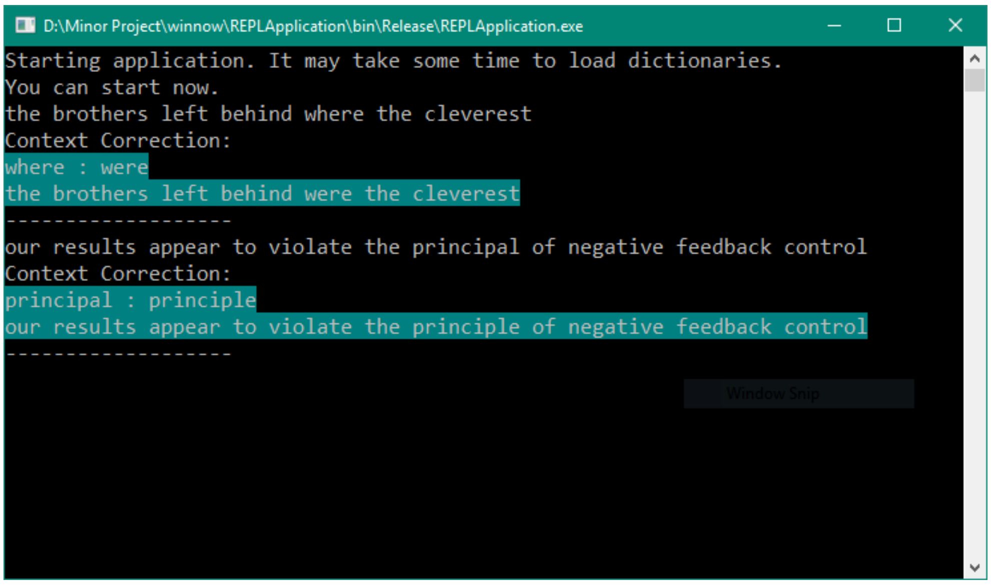
```
C:\Windows\system32\cmd.exe
Extracting Features for hour,our 15-11-2016 04:35:29 AM
Training done for weather,whether 15-11-2016 04:37:52 AM
Extracting Features for fourth,forth 15-11-2016 04:37:53 AM
Training done for fourth,forth 15-11-2016 04:38:06 AM
Extracting Features for passed,past 15-11-2016 04:38:07 AM
Extracting Features for hear,here 15-11-2016 04:38:46 AM
Training done for passed,past 15-11-2016 04:39:20 AM
Extracting Features for threw,through 15-11-2016 04:39:21 AM
Extracting Features for later,latter 15-11-2016 04:39:21 AM
Extracting Features for cite,site,sight 15-11-2016 04:39:26 AM
Training done for hour,our 15-11-2016 04:41:02 AM
Extracting Features for by,buy,bye 15-11-2016 04:41:07 AM
Training done for later,latter 15-11-2016 04:42:09 AM
Extracting Features for coarse,course 15-11-2016 04:42:10 AM
Training done for knew,new 15-11-2016 04:44:52 AM
Extracting Features for vain,vane,vein 15-11-2016 04:44:54 AM
Training done for vain,vane,vein 15-11-2016 04:45:02 AM
Extracting Features for loose,lose 15-11-2016 04:45:04 AM
Training done for cite,site,sight 15-11-2016 04:45:11 AM
Extracting Features for plaine,plane 15-11-2016 04:45:12 AM
Training done for plaine,plane 15-11-2016 04:45:26 AM
Extracting Features for principal,principle 15-11-2016 04:45:27 AM
Training done for loose,lose 15-11-2016 04:45:28 AM
Extracting Features for sea,see 15-11-2016 04:45:29 AM
Training done for threw,through 15-11-2016 04:45:35 AM
Training done for coarse,course 15-11-2016 04:45:40 AM
Extracting Features for desert,dessert 15-11-2016 04:45:42 AM
Extracting Features for to,too,two 15-11-2016 04:45:47 AM
Training done for desert,dessert 15-11-2016 04:45:52 AM
Training done for principal,principle 15-11-2016 04:46:28 AM
Training done for hear,here 15-11-2016 04:46:53 AM
Training done for sea,see 15-11-2016 04:52:15 AM
Training done for their,there 15-11-2016 05:03:01 AM
Training done for where,were 15-11-2016 05:03:02 AM
Training done for by,buy,bye 15-11-2016 05:22:01 AM
Training done for to,too,two 15-11-2016 06:53:15 AM
Extracting Features for brake,break 15-11-2016 06:53:15 AM
Training done for brake,break 15-11-2016 06:53:16 AM
feature extraction + training took 149.603767731667 Minutes
Prunning:Off
-----------------------------------------------
Test Accuracy: 94 % of 60553 test samples, took: 191 Minutes
Press any key to continue . . .
```

Figure 4.1: Training of the models

4.4.2 Testing

1. Read the feature vectors and trained model from the serialized file.
2. Read input sentences from a console / file.
3. Pass the sentences to the trained classifier model which makes its corrections, if any.
4. Also pass the sentences to other classifier modules available (Levenshtein distance etc.) and collect their corrections if any.
5. Present the list of corrections to the user as possible corrections.

When a sentence is presented, the system checks the classifiers of each confusion set, if the sentence contains a word from a confusion set, features are extracted and run against the corresponding classifier, and returns the prediction with the highest score.

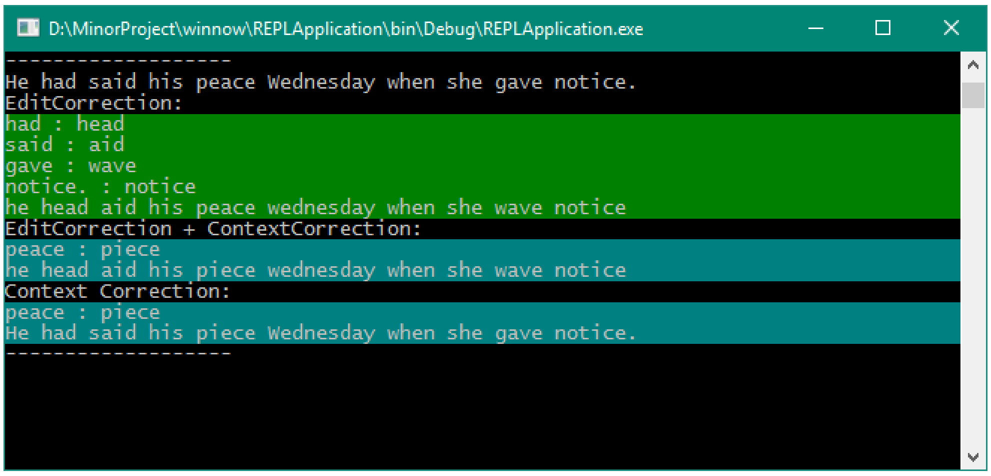


D:\Minor Project\winnow\REPLApplication\bin\Release\REPLApplication.exe

```
Starting application. It may take some time to load dictionaries.
You can start now.
the brothers left behind where the cleverest
Context Correction:
where : were
the brothers left behind were the cleverest
-------------------
our results appear to violate the principal of negative feedback control
Context Correction:
principal : principle
our results appear to violate the principle of negative feedback control
-------------------
```

Figure 4.2: Correction of a sentence using Context correction

If the target doesn't accumulate enough features (example may be when the target word falls at the end of a sentence), the *Edit Distance corrector* kicks in and tries to find the word from the dictionary or the confusion sets with the minimum *edit distance*. This helps to ensure that the predictions can be made with some heuristic even if there is not enough contextual information available.



D:\MinorProject\winnow\REPLApplication\bin\Debug\REPLApplication.exe

```
-------------------
He had said his peace Wednesday when she gave notice.
EditCorrection:
had : head
said : aid
gave : wave
notice. : notice
he head aid his peace wednesday when she wave notice
EditCorrection + ContextCorrection:
peace : piece
he head aid his piece wednesday when she wave notice
Context Correction:
peace : piece
He had said his piece Wednesday when she gave notice.
-------------------
```

Figure 4.3: Prediction of a sentence using Winnow

Results

Running time for feature extraction + training + testing took on average 59 minutes on my i7, 3.4 GHz, 8 GB RAM machine. The above results were achieved with pruning disabled, when enabled, accuracy went down and running time increased.

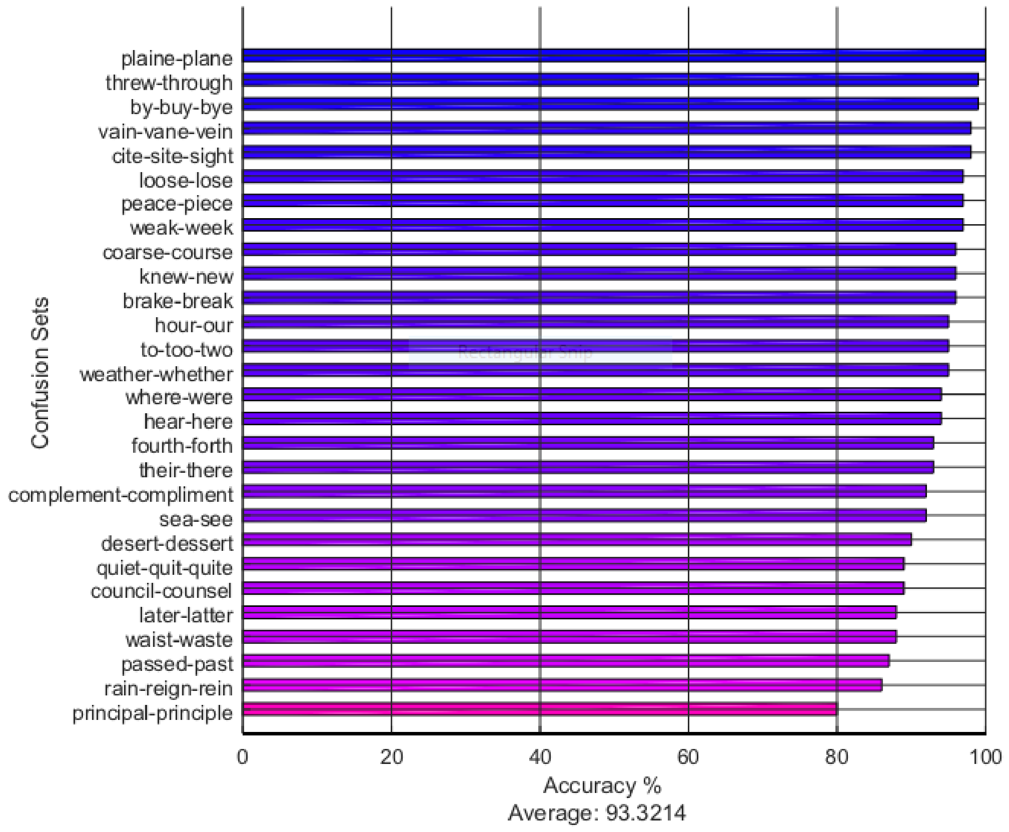


Figure 5.1: Results

5.1 Key Features of Project

1. Easily Extensible: New classification algorithms can be added easily as separate modules.
2. Very small footprint. Takes just 60MB of disk space and / or RAM.
3. Very fast algorithm. Can correct up to 20 sentences per second.

5.2 Future Scope

1. In training, for each word there's an array of classifiers, in the code I kept the number to 1, we can add more algorithms like logistic regression or even neural networks, then pick the best result using Weighted Majority Algorithm.
2. In context features extraction, we can use a dictionary API to properly stem and singularize the tokens.
3. Soundex algorithm can be used to generate more confusion sets.
4. WordNET can be used to extract 'concepts' from the sentence and try to choose the correction candidate belonging to the same 'concept'.
5. Multiple classifiers can be combined to provide better results and a more varied coverage of errors. e.g. TF-IDF, n-grams probability model etc.
6. A ranking algorithm can be used to give different weights to different classifiers.