Detecting Grammatical Errors in Text using a Ngram-based Ruleset

Manu Konchady,

Mustru Search Services,
118, RMV Extension, Stage 2, Block 1,
Bangalore, 560094. India.
mkonchady@yahoo.com

***Abstract*— Applications like word processors and other writing tools typically include a grammar checker. The purpose of a grammar checker is to identify sentences that are grammatically incorrect based on the syntax of the language. The proposed grammar checker is a rule-based system to identify sentences that are most likely to contain errors. The set of rules are automatically generated from a part of speech tagged corpus. The results from the grammar checker is a list of error sentences, error descriptions, and suggested corrections. A grammar checker for other languages can be similarly constructed, given a tagged corpus and a set of stop words.**

I. Introduction

A grammar checker verifies free-form unstructured text for grammatical correctness. In most cases, a grammar checker is part of an application, such as a word processor. In this paper, a Web-based grammar checker is implemented to verify the correctness of short essays that are submitted by students in competitive exams. An essay can vary from a single paragraph to a medium sized document made up of several pages (~ 100 Kbytes).

The earliest grammar checkers in the 80s searched for punctuation errors and a list of common error phrases. The task of performing a full blown parse of a chunk of text was either too complex or time consuming for the processors of the early PCs. Till the early 90s, grammar checkers were sold as separate packages that were installed with a word processor. The software gradually evolved from a set of simplistic tools to fairly complex products to detect grammatical mistakes beyond a standard list of common style and punctuation errors.

While a grammar checker verifies the syntax of language, a style checker compares the use of language with patterns that are not common or deprecated. A style checker may look for excessively long sentences, out-dated phrases, or the use of double negatives. We have not considered style checking in this work and have focused on syntax verification alone. Further, there is no verification of semantics. For example, the sentence - "*Colorless green ideas sleep furiously*." was coined by Noam Chomsky to illustrate that sentences with no grammatical errors can be nonsensical. Identifying such sentences requires a large knowledge corpus to verify the semantics of a sentence.

Requirements

The main purpose of a grammar checker is to help create a better document that is free of syntax errors. A document can be analyzed in its entirety or one sentence at a time. In a batch mode, the entire text of a document is scanned for errors and the results of a scan is a list of all possible errors in the text. An online grammar checker will identify errors as sentences are detected in the text. Grammar checkers can be computationally intensive and often run in the background or must be explicitly invoked.

One of the primary requirements for a grammar checker is speed. A practical grammar checker must be fast enough for interactive use in an application like a word processor. The time to analyze a sentence should be sub-second or less following an initial startup time.

The second requirement is accuracy. A grammar checker should find all possible errors and correct sentences as well. There are two types of errors. The first type of error is a false positive or an error that is detected by the grammar checker but which is not an actual error. The second type of error is an actual error that was not detected by the grammar checker (a false negative). In general, the number of false positives are minimized to avoid annoying the user of the application.

The third requirement to limit the number of correct sentences that are flagged as errors by the grammar checker, is related to the second requirement. At first, we may assume that simply setting the threshold high enough for an error should be sufficient to satisfy this requirement. However, a grammar checker with a threshold that is too high will miss a large number of legitimate errors. Therefore, the threshold should be such that the number of false positives are minimized while simultaneously reducing the number of false negatives as well. The *accuracy* parameter defined in the Evaluation section combines these two attributes in a single value, making it possible to compare grammar checkers.

Since it is difficult to set an universal threshold that is appropriate for all situations, the user can select a level of "strictness" for the grammar corrector. A higher level of strictness corresponds to more rigorous error checking.

Emustru

The grammar checker in this paper is embedded in the open source Emustru project [2]. The purpose of this project is to teach language skills, specifically spelling and writing skills. The project is aimed at high school or entry level college students who want to improve their writing skills. Spelling lists from textbooks prescribed for high school students studying the central board (CBSE) syllabus and from the Brown Corpus [1] are incorporated in Emustru.

An online Web-based essay evaluator in Emustru accepts a chunk of text written in response to an essay type question, that elicits the opinion of the writer regarding an issue or topic. The essay evaluator uses the number of grammatical errors in addition to other parameters such as the use of discourse words, organization, and the number of spelling errors in the text to assign an overall evaluation score.

The essay evaluator returns a score and a category for the essay along with a list of parameters computed from the text of the essay. The grammar checker returns results for each sentence extracted from the text. A sentence that is incorrect is flagged and a description explaining the error along with a suggestion is given.

Sentence: My **farther is** fixing the computer.
Description: The tag an adverb, comparative is not usually followed by **is**
Suggestion: Refer to **farther** and **is**

The words in the sentence that are incorrect are highlighted. The description and suggestion are generated automatically based on the type and location of the error. The checker marks words in the sentence that is part of an error and subsequent errors due to the same words are ignored. Therefore, some sentences may need to be corrected more than once.

In Section III, the design of the Emustru grammar checker is explained. A ruleset is automatically generated from a tagged corpus and used to detect potential errors. This method is purely statistical and will be inaccurate when the tagged corpus does not cover all possible syntax patterns or if the tagged corpus contains mis-tagged tokens. Further, since the grammar checker uses a trained POS tagger, the accuracy of the checker is constrained by the correctness of the POS tags assigned to individual tokens of sentences.

Despite these inherent problems with a statistically-based grammar checker, the results are comparable with the grammar checker used in the popular Microsoft Word™ word processor (see Section V). A sample corpus of 100 sentences made up of 70 correct and 30 incorrect sentences was used in the evaluation. The accuracy of the grammar checker can be adjusted using a likelihood parameter. The grammar checker has also been evaluated using the standard Information Retrieval *recall* and *precision* parameters. Finally, some improvements and the results are discussed in the conclusion section.

II. Prior Work

Grammar checkers first divide a chunk of text into a set of sentences before detecting any errors. A checker then works on individual sentences from the list of sentences. Two tasks that are necessary in all grammar checkers are sentence detection and part of speech (POS) tagging. Therefore, this dependency limits the accuracy of any grammar checker to the combined accuracy of the sentence detector and POS tagger. Sentence detectors have fairly high precision rates (above 95%) for text that is well-written such as newswire articles, essays, or books. POS taggers also have high accuracy rates (above 90%), but have a dependency on the genre of text used to train the tagger.

Two methods to detect grammatical errors in a sentence have been popular. The first method is to generate a complete parse tree of a sentence to identify errors. A sentence is parsed into a tree like structure that identifies a part of speech for every word. The detector will generate parse trees from sentences that are syntactically correct. An error sentence will either fail during a parse or be parsed into an error tree.

One problem with this approach is that the parser must know the entire grammar of the language and be able to analyze all types of text written using the language. Another problem is that some sentences cannot be parsed into a single tree and there are natural ambiguities that cannot be resolved by a parser. A grammar checker in the open source word processor, AbiWord uses a parser from Carnegie Mellon University to find grammatical errors.

The second method is to use a rule-based checker that detects sequences of text that do not appear to be normal. Rule-based systems have been successfully used in other NLP problems such as POS tagging [4]. Rule-based systems have some advantages over other methods to detect errors. An initial set of rules can be improved over time to cover a larger number of errors. Rules can be tweaked to find specific errors.

A. Manual Rule-based Systems

Rules that are manually added can be made very descriptive with appropriate suggestions to correct errors. LanguageTool [3] developed by Daniel Naber is a rule-based grammar checker used in OpenOffice Writer and other tools. It uses a set of XML tagged rules that are loaded in the checker and evaluated against word and tag sequence in a sentence. A rule to identify a typo is shown below.

```
<rule id="THERE_EXITS" name="Possible typo: 'There exits' (There exists)">
  <pattern mark_from="1">
```

```
    <token>there</token>
    <token>exits</token>
  </pattern>
   <message>Possible typo. Did you mean <suggestion>
exists </suggestion>?
   </message>
  <example correction="exists" type="incorrect">
    There <marker>exits</marker> a distinct possibility.
  </example>
  <example type="correct">
   Then there exists a distinct possibility.</example>
 </rule>
```

Every rule begins with *id* and *name* attributes. The *id* is a short form name for the rule and the *name* attribute is a more descriptive text that describes the use of the rule. The pattern tags describe the sequence of tokens that the checker should find in a sentence, before firing this particular rule. In this example, the two consecutive tokens – *there* and *exits* define the pattern.

Once a rule is fired, a message and a correction is generated. Since rules are manually generated in LanguageTool, the error description and correction are very precise. The section of text from the sentence that matches the pattern can be highlighted to indicate the location of the error in the sentence.

A rule with tokens in a pattern is quite specific, since the identical tokens must occur in the matching sentence, in the same order as the tokens in the pattern. More general rules may use POS tags instead of specific tokens in the pattern. For example, a rule may define a pattern where a noun tag follows an adjective tag. This particular order of tags is rare in English and is a potential error.

LanguageTool uses many hundreds of such rules to find grammatical errors in a sentence. Some of the patterns of these rules include regular expression-like syntax to match a broader variety of tag and token sequences. Although, LanguageTool is a very precise grammar checker, there are two drawbacks. One, the manual maintenance of several hundreds of grammar rules is quite tedious. It has become a little simpler to collaboratively manage large rule sets with the use of Web-based tools. Two, the number of rules to cover a majority of the grammatical errors is much larger. Therefore, the recall of LanguageTool is relatively low. Finally, each language requires a separate set of manually generated rules.

Other rule-based checkers include EasyEnglish from IBM Inc. and a Constitutent Likelihood Automatic Word-tagging System (CLAWS) probabilistic tagger to identify errors. The well known grammar checker used in Microsoft Word is closed source and many of the other grammar checkers are similarly not available to the public. The design of the Emustru grammar checker is based on a probabilistic tagger suggested by Atwell [5]. Rules are generated automatically from a tagged corpus and errors are identified when low-frequency tag sequences are observed in a sentence. The assumption is that a frequent tag sequence in a tagged corpus that has been validated is correct.

B. Automatic Rule-based Systems

Grammar checkers based on automatically generated rule sets have been shown to have reasonable accuracy [6,7] to be used in applications such as Essay Evaluation. The automated grammatical error detection system called ALEK is part of a suite of tools being developed by ETS Technologies, Inc. to provide students learning writing with diagnostic feedback. A student writes an essay that is automatically evaluated and returned with a list of errors and suggestions. Among the types of errors detected are spelling and grammatical errors.

The ALEK grammar checker is built from a large training corpus of approximately 30 million words. Corpora such as CLAWS and the Brown corpus, characterize language usage that has been proofread and is presumed to be correct. The text from these corpora is viewed as *positive* evidence that is used to build a statistical language model. The correctness of a sentence is verified by comparing the frequencies of chunks of text from the test sentence with similar or equivalent chunks in the generated language model.

A collection of ill-formed sentences consitutes *negative* evidence for a language model. Text chunks from a sentence that closely match a language model built from negative evidence are strong indicators of potential errors. However, it is harder to build a corpus of all possible errors in a language. The number and types of errors that can be generated are very large. Consider a four word sentence.

My name is Ganesh.

There are 4! or 24 ways of arranging the words of this particular sentence, of which only one is legitimate. Other sources of errors include missing words or added words that make ill-formed sentences. The construction of a corpus made up of negative evidence is time consuming and expensive. Therefore like ALEK, the Emustru grammar checker uses positive evidence alone to build a language model.

Text is preprocessed (see Preprocessing section) before evaluation. A grammar check consists of comparing observed and expected frequencies of words / POS tag combinations. This same method is used to identify phrases such as "*New Delhi*" or "*strong tea*" in text. Bigram tokens such as these are seen more frequently than by chance alone and therefore have a higher likelihood of occurrence.

Consider the phrase "*New York*" in the Brown corpus. The probability of observing the word "*York*" when it is preceded by the word "*New*" is 0.56, while the probability of observing "*York*" when it is preceded by any word except "*New*" is

0.00001. These probabilities along with individual word counts are used to find the likelihood that two words are dependent.

The log-likelihood measure [7] is suggested when the observed data counts are sparse. An alternate mutual information measure compares the expected relative frequency of a bigram in the corpus to the expected relative frequency assuming the bigram is independent.

$$MI = \log\left(\frac{p(name-is)}{p(name)\times p(is)}\right)$$

where *p(name-is)* is the probability of the bigram "*name is*" and the denominator is the product of the unigram probabilities of "*name"* and "*is"*. Both the mutual information and log-likelihood measures have been used in the Emustru grammar checker. The log-likelihood measure is used when the number of occurrences of one of the words is less than 100 (in the Brown corpus).

A generated statistical language model is a large collection of word/tag pairs The occurrence of words and tags in text is not independently distributed, but instead has an inherent association built in the usage patterns that are observed in text. For example, we would expect to see the phrase "*name is*" more often than the phrase "*is name*". A rule would assign a much higher likelihood for the phrase "*name is*" than the phrase "*is name*". The design for the ruleset used in the Emustru grammar checker is based on a large number of these types of observations.

III. Design

The design of the Emustru grammar checker is made up of three steps. The first preprocessing step is common to most grammar checkers. Raw text is filtered and converted to a list of sentences. Text extracted from files often contains titles, lists, and other text segments that do not form complete sentences. The filter removes text segments that are not recognizable as sentences. The text of each extracted sentence is divided into two lists of POS tags and tokens. Every token of a sentence has a corresponding POS tag. The lists of tags and tokens for a sentence are passed to the checker.

The second step is to generate a rule set that will be used by the checker. In this step, four tables consisting of several thousand rules are automatically generated from a tagged corpus and lists of stop words and stop tags. The final step is the application of the generated rules to detect errors. The lists of tokens and tags are analyzed for deviations from expected patterns seen in the corpus. Sequences of tags and tokens are evaluated against rules from four different tables for potential errors. The first error in a tag / token sequence that may have multiple errors is returned from the grammar checker. This limits the total number of errors per sentence.

A. Preprocessing

A pipeline design is used in the Emustru grammar checker. The raw text is first filtered and converted into a stream of sentences. The sentence extractor from LingPipe is used to extract sentences from the text (see Figure 1). The sentence extractor uses a heuristic method to locate sentence boundaries. The minimum and maximum lengths of sentences are set to 50 and 500 characters respectively. Most English sentences end with a sentence terminator character, such as a period, comma, or a exclamation. These characters are usually followed by a space and the first word of the following sentence or the end of the text.

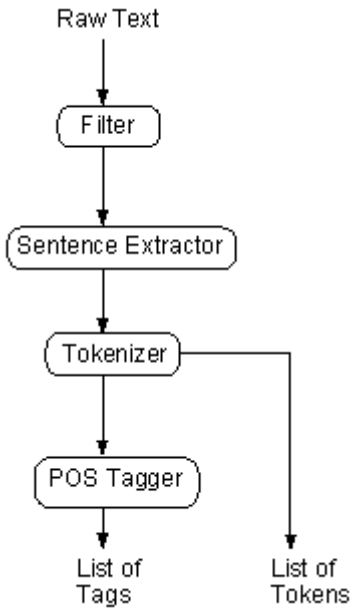

Fig. 1 Extracting lists of tokens and POS Tags

The sentence extractor will fail to extract sentences that do not separate the sentence terminator from the first word of the next sentence. Instead a complex token such as *abc.com* or an abbreviation will be assumed. A POS tagger accepts a list of tokens from a sentence and assigns a POS tag to each token. The output from the preprocessing step is a list of tokens and associated tags per extracted sentence. Most of the tokens in a sentence can be extracted by simply splitting the sentence string when a whitespace is observed. Although this works for most tokens, some tokens such as *I'll* or *won't* are converted to their expanded versions "*I will*" and "*will not*". Other tokens such as *out-of-date* and *Sourceforge.net* are not split into two or more tokens. Tokens that contain periods such *Mr.* or *U.S.* are retained as is.

B. Creating a Rule Set

The rule set used in the grammar checker is a collection of four database tables. A tagged corpus and lists of stop words and tags are used to build the set of rule database tables (see Figure 2). The rule set is created once before the grammar checker can be applied. A modified rule table must be re-loaded in the database to take effect.

Unigrams

The first table is the unigram table. This table contains the most common tags for words in the dictionary. A POS tag *y* that was assigned to a word *x* in fewer than 5% of all cases in the tagger corpus is noted in a rule for *x*. Any sentence that contains the word *x* tagged with *y* is considered a potential error by the checker. The types of errors detected are pairs of words that are used incorrectly such *affect* and *effect* or *then* and *than*. For example, the probability of finding the word *affect* used as a noun was less than 3% in the Brown corpus. The unigram rule for the word *affect* will detect the erroneous use of the word in the sentence below.

We submit that this is a most desirable ***affect*** *of the laws and one of its principal aims.*

The grammar checker returns the following description - "The word *affect* is not usually used as a noun, singular, common" and the suggestion - "Refer to *affect*, did you mean *effect*". There are numerous other pairs of such words that are often mixed up, such as bare / bear, accept / except, and loose / lose.

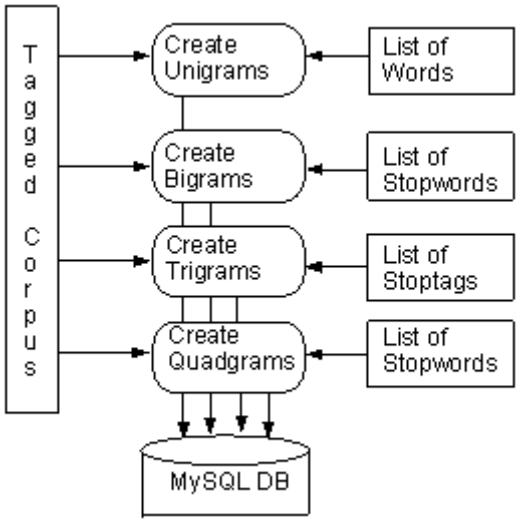

Fig. 2 Create a Ruleset made up of Four Database Tables

Bigrams

The bigram tag table is constructed by observing tag sequences in the corpus and computing a likelihood measure for each tag sequence. Consider the erroneous sentence – "My father fixing the computer.". The tag sequences extracted from this sentence and their likelihoods are shown in Table 1. The *START* and *END* tags are added to the beginning and the end of the sentence respectively.

TABLE I BIGRAM TAG SEQUENCES FOR AN ERRONEOUS SENTENCE.

Token	***Tag Sequence***	***Likelihood***	***Error***
My	START-PP$	0.33	No
father	PP$-NN	1.93	No
fixing	NN-VBG	-1.11	Yes
the	VBG-AT	0.71	No
computer	AT-NN	1.90	No
.	NN-.	1.32	No

All the tag sequences in Table 1 have positive likelihoods with the exception of the NN-VBG tag sequence. The likelihood of this tag sequence is the likelihood of a verb or present participle following a noun. It is negative since a present participle is usually preceded by a present tense verb such as *is*. These types of errors are found in sequences of bigram tags.

Other types of bigram sequences include tag-word and word-tag sequences. Words found in text are separated into two sets – open class and closed class words. The open class set contains mainly nouns, adjectives, verbs, and adverbs. These words are numerous and together are found often in text. The individual frequency of a noun or adjective is typically small compared to the frequency of a closed class word. Conjunctions, prepositions, and articles are fewer in number but occur often in text. Golding [9] showed that it is possible to build context models for word usage to detect errors. The context of a word *x* that does not match the context defined in the bigram table for *x* is a potential error.

The words that are used most frequently in the tagged corpus are selected in a stop list that includes words such as – *the*, *and*, *of*, and *did*. Consider tag-word rules for the word *the* that model the context of tags before the stop word. An adjective is rarely seen before the word *the. T*he rule with the "*JJ-the*" context will detect an error in the sentence – "*Can you make large the photo?*". Similarly in the sentence - "*The goal to find was who attended*." the word-tag rule for the "*was-WPS*" context detects an error in the word sequence "*was who*". All three types of sequence rules – tag-tag, tag-word, and word-tag are used to detect bigram error sequences in sentences.

Trigrams

The use of trigrams to model word / tag usage requires a very large corpus. Consider the Brown corpus with roughly 100 POS tags. The maximum number of trigram tag sequences that can be generated is one million. The number of words in

the Brown corpus is one million and is clearly not sufficient to model the usage patterns of all possible trigram tag sequences.

Instead, the problem is limited to modelling a much smaller set of trigram tag sequences. The modelled set of tag sequences represents tags that are frequently found in grammatically incorrect sentences. For example in the sentence - "*I did not wanted to clean the room.*", the verb *want* is used in the wrong tense. The sentence is correct if the present tense of the word is used instead of the past tense. We can collect pairs of such tags that are interchanged in grammatically incorrect sentences. These tags form a stop tag list that have one or more replacement tags that may fix the error. The grammar checker returns the following description and suggestion for the incorrect sentence above.

Description: The fragment ***not wanted to*** is rare.

Suggestion: Possible agreement error: Replace *wanted* with verb, base: uninflected present ...

The detector uses the tags before and after the stop tag to build the context of the given sentence. The likelihood of the tag sequence is extracted from the database table and compared with the likelihood of another tag sequence that replaces the stop tag with a substitute tag. An error is generated when the likelihood of the tag sequence with the substitute tag is much higher than the likelihood with the original tag. Consider another incorrect sentence - "*She come to college late every day.*". The present tense of the verb *come* is used instead of the past tense. Here, the grammar checker returns -

Description: The fragment ***She come to*** is rare.

Suggestion: Possible agreement error: Replace *come* with verb, past tense.

The purpose of using trigrams is to identify errors that the bigram tables fail to detect. For example, in the first sentence the token sequences "*not wanted"* and "*wanted to"* are both legitimate token sequences independently. However, the combined tag sequence "*not wanted to"* is rare and is a potential error. The replacement of the past tense tag with the present tense tag produces a tag sequence that is more likely than the original tag sequence. The checker generates errors for cases where the replacement tag creates a more likely tag sequence.

This is not a fool-proof method, since the best possible replacement tag cannot be predicted beforehand in a stop tag list. An attempt is made to find the pairs of tags that are most often mixed up in an error corpus. A stop tag may also have more than one replacement tag. In such situations, the most likely replacement tag is selected to compare with the liklelihood of the original tag. Finally, a corpus larger than the one million word Brown corpus is needed to accurately model trigram tag sequences.

Quadgrams

An extremely large corpus would be needed to model all possible quadram tag sequences for the same reasons as the trigram tag sequences mentioned earlier. The number of possible quadgram sequences is very large and accurately modelling the usage patterns of such a huge number of sequences would require a corpus that is not currently available. The space and time required to build a quadgram model would be correspondingly large.

The quadgram model is simplified to identify specific words that are used in the wrong context. Quadgram sequences are constructed for a set of stop words. These stop words represent pairs of words like is / are, was / were, and there / their. Consider the sentence - "*A herd of horses are better than a flock of sheep.*". The grammar checker returns with the following description and suggestion:

Description: The fragment ***better than a*** is not usually preceded by ***are***

Suggestion:Possible agreement error: Replace ***are*** with ***is***

The checker begins by constructing two quadgrams when a stop word is observed in the sentence. For example, in the sentence above, the two quadgrams - "*herd of horses are*" and "*are better than a*" are generated when the word *are* is seen. All the words in the quadgrams are replaced by their corresponding tags with the exception of the stop word (*are*). The use of tags instead of the specific words themselves, makes the quadgrams more general and easier to model with a smaller corpus. The likelihood of both quadgrams with the given stop word *are,* is evaluated using the Quadgram database table.

The stop word *are* is replaced with *is* in both quadgrams and the likelihood of the modified quadgrams is extracted from the database table as before. If the likelihood of the modified quadgram substantially exceeds the likelihood of the original quadgram, then the checker generates an error.

The quadgram model is subject to the same types of problems as the trigram model. We need to know beforehand words that are frequently used incorrectly and the appropriate replacement word. The type of words included in the quadgram stop word list are seen in subject-verb agreement errors. The stop word list used in the Emustru grammar checker is made up of a few of these types of words. The size of the Brown corpus may not be sufficient to build an accurate model for these quadgrams.

Long distance word dependencies in sentences are not detected by the bigram or trigram table look ups. Such

dependencies occur when the subject and verb are separated by one or more words. making it difficult for the bigram or trigram checker to detect the low likelihood of a plural form of a verb used with a singular form of a noun. The quadgram table models some of these long distance word dependencies that are missed in the earlier steps.

Error Model

The spelling error model has been adapted to describe a grammar error model. In the spelling error model, four modification functions that operate at the character level are used to correct the spelling of a word. For example, the transpose function will interchange the letters *i* and *e* in the mis-spelled word *recieve* to create the correct word *receive*. One of more of these four functions can be used to correct any mis-spelled word. The edit distance is a measure of the number of times a modification function needs to be applied to transform a mis-spelled word into the correct word. The grammar error model uses the same functions as the spelling error model, except that the modifications for the grammar error model operate at the word level (see Table 2).

TABLE II TYPES OF GRAMMAR ERRORS.

Error	***Sentence***
Delete	Why did the chicken cross road?
Insert	Who is the the chicken?
Transpose	The chicken's food is from made soya.
Modify	The number of chickens were large.

For example, the delete error in the first row of Table 2 is corrected by applying the insert function that adds the word *the* between the words *cross* and *road*. Similarly, the transpose function transforms the word sequence "*from made*" to "*made from*" in the third row of Table 2. The rules in the ngram tables that detect these errors is shown in Table 3.

TABLE III DATABASE TABLES USED TO CORRECT ERRORS IN TABLE II.

Error	DB Table	Message
Delete	Bigram (tag-tag)	A noun is not usually followed by a noun (refer to *cross* and *road*).
Insert	Bigram (word-tag)	The token *the* is not usually followed by an article (refer to *the* and *the*).
Transpose	Bigram (word-tag)	The token *is* is not usually followed by a preposition (refer to *is* and *from*).
Modify	Quadgram (tag-word)	The fragment *number of chickens* is not usually followed by *were* (Possible agreement error: Replace *were* with *was*)

This is a simplistic error model and does not make distinctions between errors such as subject-verb agreements, run-ons, and other grammatical mistakes. Although the error model is unsophisticated, the descriptions and suggestions are usually good enough for the user to correct an error. The part of the sentence that contributed to the error is highlighted and the automatically generated description explains why the tag(s) or token were not appropriate in the sentence.

C. Applying a Rule Set

The input to the grammar checker is a list of tokens and tags along with the stop lists for each of ngram checks. The four ngram database tables are checked in order starting from the Unigram to the Quadgram table (see Figure 3). The output from the grammar checker is a list of possible errors.

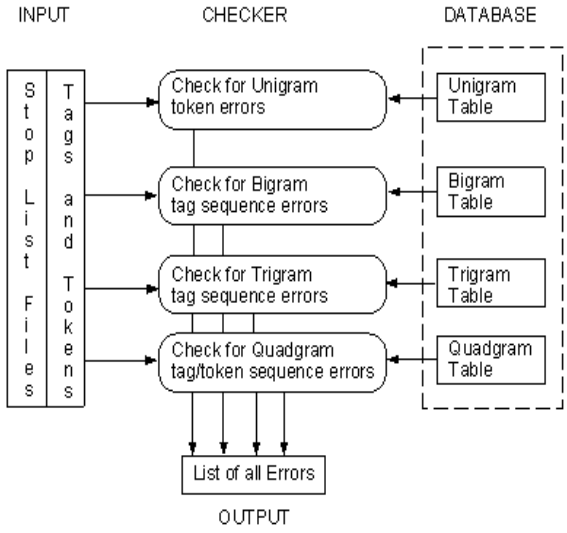

Fig. 3 Applying a Ngram-based Rule Set to Detect Errors

Each of the errors returned contains a description and a suggestion. The text for these fields is automatically generated and therefore not as precise as a message from a manually generated rule. An unigram error states that a word is rarely used with the assigned POS. A bigram error mentions that either a word is not usually followed by a tag or two assigned tags are rarely seen together. The trigram and quadgram errors suggest some type of agreement error and propose alternate tags or alternate words to correct a sentence.

IV. IMPLEMENTATION

The Emustru grammar checker has been implemented in Java using a number of open source tools that include dictionaries, a tagged corpus, and the Google Web Toolkit (GWT) (see Acknowledgments section). The Web-based implementation uses the GWT to handle client requests and display the list of results (see Figure 4).

The client makes a request via a browser to a Php script on the server. The Php script accepts a chunk of text passed by the client and generates a temporary file containing the passed text. The grammar checker is invoked by the Php script as a jar file and the name of the temporary file in an argument list.

The results from the grammar checker are sent back to the Php script in either a JSON or XML format. The Php script forwards the string to the client. Finally, the client displays the contents of the results in a table on the browser.

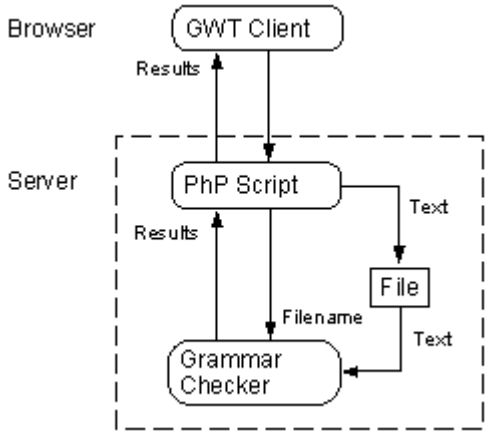

Fig. 4 Implementation of the Grammar Checker

The grammar checker is embedded in an essay evaluator. The results from the checker are shown in a tabbed window along with other evaluation measures such as the vocabulary usage, and spelling errors.

V. Evaluation

The Emustru grammar checker has been evaluated using a small corpus of 100 annotated sentences. A fraction (70%) of the sentences are grammatically correct and the remainder have one or more errors. This is a small corpus and a large scale evaluation would use a wider variety of sentences and errors to test the checker.

The corpus to test the grammar checker is a collection of *e* sentences, out of which *a* sentences ($a < e$) are grammatically incorrect. The grammar checker was run against the set of *e* sentences to detect errors. The results of the test are shown in Table 4.

TABLE IV Grammar checker Evaluation

		Actual Errors	
		Yes	***No***
Flagged Errors	***Yes***	*a*	*b*
	No	*c*	*d*

where $e = a + b + c + d$. The value of *a* is the number of errors that were actually errors and correctly flagged by the checker. The value of *b* is the number of errors that were not found by the checker. The value of *c* is the number of errors that were wrongly identified by the checker. Finally, *d* is the number of correct sentences that was assigned zero errors by the checker.

Most of the sentences in the test corpus have been selected from various news articles on the Web. News articles from reputed sources have been proofread and can be presumed to be error-free. A sample of sentences from news articles on different topics were selected for the set of correct sentences. The set of incorrect sentences were generated from the most common types of grammatical errors mentioned on the Web.

Notice, this error corpus is not annotated to mention a particular type of grammatical error. Instead, a sentence is merely defined as correct or incorrect. Therefore, there is no verification of error type detected to match a particular grammatical error. Any error detected in a sentence that is invalid is consided as a correctly flagged error and tabulated in the value of *a* in Table 4. Currently, there is no standard corpus for grammatical error detection and no standard format to define a syntax error.

Recall and *precision* are two standard evaluation parameters used in Information Retrieval to evaluate the performance of search engines. In this context, we can define recall as the value $a / (a + c)$ and precision as $a / (a + b)$. Typically, recall and precision are inversely related. i.e. high recall is associated with low precision and vice versa. Consider an extreme case where $a = e$ and every sentence in the test corpus is flagged as an error. Recall will be 1.0 or the maximum since the value of *c*, the number of missed errors will be zero. However, precision will be low since the value of *b*, the number of wrongly detected errors will be high.

Conversely, the checker may flag a very small fraction of errors that are obvious. In this case, the value of *a* will be very small and correspondingly the value of *b* will be zero leading to precision of 1.0. However the value of *c*, the number of actual errors that were not flagged by the checker will be high making recall low.

This problem of balancing recall and precision is fairly common in other NLP tasks such as entity extraction and sentiment analysis. The implementation in this paper attempts to maximize precision with reasonable recall. In other words, flagging an excessive number of errors is more annoying to the user than allowing a few undetected errors.

We can vary the threshold higher or lower to detect fewer or more errors respectively. The threshold should be high enough to minimize the number of false positives, i.e. the sentences the checkers believes are errors, but are actually correct. In Table 4, this means that the checker should minimize the value of *b*. Recall is controlled to a lesser extent by minimizing the value of *c*, i.e. finding as many of the actual errors as possible.

A third evaluation parameter is *accuracy*. This parameter combines all the results in Table 4 into a single value. It is defined as $(a + d) / (a + b + c + d)$. Accuracy measures the number of error sentences and correct sentences detected relative to the total number of sentences.

Table 5 contains four sample sentences from the test corpus of 100 sentences, two sentences each from the *correct* and *error* categories. These sentences are similar in style to the other sentences in the corpus. There is a large number of grammatical error types and the test corpus covers some of the common errors. The types of errors covered include subject-verb agreement and the incorrect location of words in a sentence.

TABLE VSAMPLE CORRECT AND ERROR SENTENCES

Category	Sentence
Error	I did good in this course.
Error	Their is a major problem with this paper.
Correct	It'll recover this year after the temporary adjustment.
Correct	Some people survive much longer based on the tumor's subtype, size, whether it has spread and the patient's age.

A test of the Emustru grammar checker with a likelihood threshold of 6.5 gave an accuracy measure of 0.81 with the values of 15, 4, 15, and 66 for the parameters *a*, *b*, *c*, and *d* respectively of Table 4. The recall and precision measures were evaluated using the same error corpus.

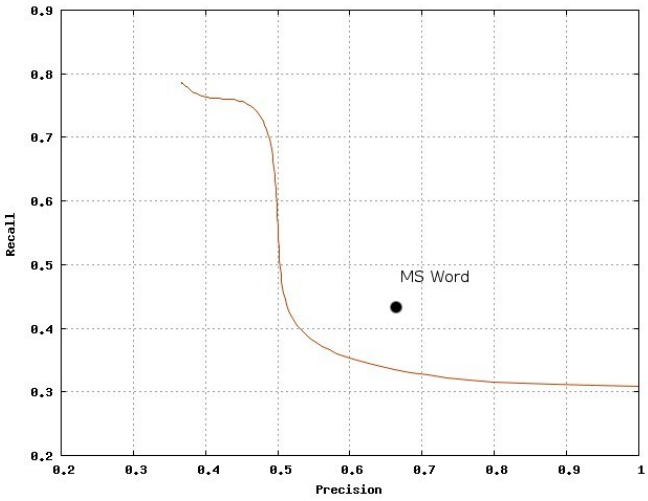

Fig. 5 Recall-Precision Plot for the Corpus of 100 sentences.

Figure 5 shows a fairly typical recall-precision plot that is observed in other information retrieval applications. When recall is high, precision is low and vice versa. The recall and precision of the MS Word grammar checker for the same error corpus was 0.433 and 0.684 respectively. The precision of the MS Word grammar checker is set high enough to ensure that very few false positives will be reported.

The accuracy of the Emustru grammar checker was varied by altering the likelihood parameter (see Figure 6). The likelihood was roughly proportional to the accuracy till a threshold of about 6.0 and thereafter the accuracy was relatively constant. We would expect low accuracy when the likelihood is low since a large number of false positives will be reported. The recall for low likelihoods is close to 1.0, while precision is much lower at 0.3. At higher likelihoods, the recall falls to about 0.5 and the precision rises up to about 0.79. The interface to the grammar checker on the client allows the user to control the likelihood parameter by adjusting the grammar level from *very strict* to *liberal*. The highest accuracy with the Emustru grammar checker of 0.81 was a slight improvement over the accuracy of 0.77 with MS Word.

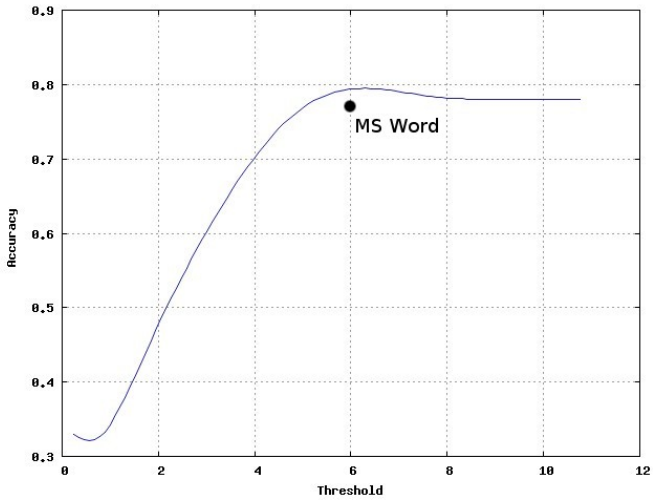

Fig. 6 Accuracy-Threshold Plot for the Corpus of 100 sentences.

There are several reasons why the accuracy of the grammar checker cannot be tweaked by adjusting the likelihood alone. The first possible source of errors is a wrong sentence boundary detection. Tokens from a sentence that is broken or a combined sentence will be harder to tag accurately. However, most sentence boundary detectors are very accurate, if the text passed is filtered to remove text fragments such as titles and table text. A second reason for low accuracy is the assignment of the wrong POS tag. This happens in roughly 5% of all tokens and therefore the grammar checker cannot possibly make an accurate judgement of a grammatical error. Finally, the tagged corpus used to build the ngram Rule sets may not accurately represent language usage patterns.

Performance

A majority of the time to detect errors is spent running SQL statements to search the four database tables. Roughly 1000 SQL select requests were required to check the error corpus of 100 sentences (2020 words). Roughly 50% and 25% of the SQL statements were lookups of the bigram and unigram tables respectively. The remaining 25% of the SQL statements were primarily lookups of the trigram table. The time to run a SQL select statement on an Intel P4 Dual-core machine with 1 Gbyte of RAM is a few milliseconds or less. Therefore, the time to analyze a sentence is roughly 30 milliseconds or less. The time to initially load the grammar checker is significant. A Hidden Markov Model-based POS tagger that is used to

assign tags to tokens is read from a file. The time to read the 6 Mbyte POS tagger file at startup time is roughly 1.25 seconds. The size of the database is shown in Table 6.

TABLE VI NUMBER OF ROWS IN NGRAM RULE SET TABLES

Table	***No. of Rows***
en_unigrams	18,379
en_bigrams	10,463
en_trigrams	19,133
en_quadgrams	17,080
Total	65,055

VI. CONCLUSIONS

The design and implementation of an open source grammar checker has been described. A statistically-based grammar checker for English has been shown to produce reasonable results compared with another grammar checker on a popular word processor. The ngram-based ruleset used to detect grammatical errors is generated automatically from a tagged corpus. The accuracy of the grammar checker can be controlled by varying a threshold lower or higher to find more of fewer errors.

The grammar checker will be evaluated with a larger test corpus to generate a more precise accuracy measure. It is feasible to use the same design to check the grammar of a different language. The main requirements are a tagged corpus , a POS tagger for the language, a set of stop tags, and a set of stop words. The generated ruleset is used to find grammatical errors in the same way the checker was used to find errors in the English language.

VII. ACKNOWLEDGMENTS

This work has been accepted as a project funded for a short term Sarai FLOSS fellowship in 2008-09 (http://www.sarai.net/). The LAMP (Linux, Apache, MySQL, and Php) platform has been used to build a Web-based implementation of the grammar checker. The browser client has been built using the Google Web Toolkit. The WordNet and Unix dictionaries have been used as the word lists to evaluate tokens. The API from LingPipe (http://www.alias-i.com/) has been used to detect sentences. The Brown Corpus [1] was used to build the set of ngram rules.

VIII. REFERENCES

[1] http://www.sscnet.ucla.edu/issr/da/index/techinfo/M0911.htm , The Brown Corpus.

[2] http://emustru.sf.net/, The Emustru Project.

[3] http://www.languagetool.org/, The LanguageTool multi-lingual grammar checker.

[4] E. Brill: A Simple Rule-Based Part of Speech Tagger, Proceedings of the Third Conference on Applied Natural Language Processing, Trento, Italy, 1992.

[5] E. Atwell, S. Elliot: Dealing with ill-formed English Text, The Computational Analysis of English, Longman Publishers, 1987.

[6] C. Leacock, M. Chodorow, Automatic Grammatical Error Detection, Automated Essay Scoring: A Cross-Disciplinary Approach, Lawrence Erlbaum Associates Publishers, 2003.

[7] Y. Attali, J. Burstein: Automated Essay Scoring with e-rater V.2, The Journal of Technology and Assessment, Vol. 4, No. 3, Feb 2006.

[8] C.D. Manning and H.Schutze, Foundations of Statistical Natural Language Processing, MIT Press, 1999.

[9] A. Golding, A Bayesian hybrid for context sensitive spelling correction, Proceedings of the Third Workshop on Very Large Corpora, pp 39-53. 1995.