AutoEval and Missplel: Two Generic Tools for Automatic Evaluation

Johnny Bigert, Linus Ericson, Antoine Solis
Department of Numerical Analysis and Computer Science
Royal Institute of Technology, Sweden
Contact: `johnny@kth.se`

Abstract

We describe two freeware programs for automatic evaluation. The first, AutoEval, greatly simplifies the data gathering, processing and counting often involved in an evaluation. To this end, AutoEval includes a simple and powerful script language to describe the evaluation task to be carried out. The second program is called Missplel. It introduces human-like spelling and grammar errors into text. A typical application is evaluation of NLP system performance on noisy input, in order to establish the robustness of the system. An evaluation of the quality of the produced errors has also been conducted.

1 Introduction

Manual evaluation of NLP systems is time-consuming and tedious. When assessing the overall performance of an NLP system, we are also concerned with the performance of the individual components. Many components will imply many evaluations. Furthermore, during the development cycle of a system, the evaluations may have to be repeated a large number of times. Sometimes, a single modification of a single component may be detrimental to overall system performance. Facing the possibility of numerous evaluations per component, we realize that manual evaluation will be very demanding.

Automatic evaluation is often a good complement to manual evaluation. Naturally, post-processing of manual evaluations, such as counting the number of correct answers, is suitable for automation. Implementation of such repetitive and monotonous tasks is carried out in evaluation of almost all NLP systems. To support the realization of these evaluations, we have constructed a program for automatic evaluation called AutoEval. This software handles all parts frequent in evaluation, such as input and output file handling and data storage, and further simplifies the data processing by providing a simple but powerful script language.

Automatic evaluation is not limited to the gathering and processing of data. Truly automatic evaluation does not require manual intervention. To this end, we have developed another program, called Missplel, which introduces human-like errors into correct text. Using this, the performance of an NLP system can be measured under the strain of ill-formed input. The system's ability to cope with noisy input is a measure of its robustness. Missplel has also been successfully used for automatic evaluation of spelling and grammar checkers.

Both programs are freeware and the source code is available from the web site (Bigert, 2003).

2 AutoEval

Evaluation is an integral part of NLP system development. Normally, the system consists of several components, where the performance of each component directly influences the performance of the overall system. Thus, the performance of the components needs to be evaluated. All evaluation has several parts in common: data

Script example:

```
(1)   field(file, "\t", "\n", :word, :tag);
(2)   ++$tot;
(3)   ++$(:tag);
```

Explanation:

`word, tag` are variables holding a word, e.g. *car* and a tag such as *NN1*.

`field` is a function that splits a file into tokens. Here, tokens are separated with a tab (`\t`) and tokens are read until a newline (`\n`) is found. The first token found is assigned to the variable `word` and the second token is assigned to `tag`.

`$` retrieves a counter (integer) variable

`++` increments a counter by one

`:tag` retrieves the content of the string variable `tag`

Figure 1: AutoEval *example. If the current input line reads* 'car NN1'*, the number of rows will be incremented by one at line (2) and the counter named* `NN1` *(which is the content of* `tag`*) will be incremented by one at line (3).*

input, data storage and processing and finally, data output. To simplify evaluation of numerous sources of NLP data, we have constructed a highly generic evaluation program, named AutoEval. The strength of AutoEval is exactly the points given above: simple input reading, automatic data storage, powerful processing of data using an extendible script language, as well as easy output of data.

2.1 Related work

Several projects have been devoted to NLP system evaluation, such as the EAGLES project (King and others, 1995), the ELSE project (Rajman and others, 1999) and the DiET project (Netter and others, 1998). Most of the evaluation projects deal mainly with evaluation methodology, even though evaluation software has often been developed to apply the methodology. For example, a PoS tag test bed was developed in the ELSE project and a spelling checker test bed was developed in the DiET project. A general tool as AutoEval would have greatly simplified the implementation of such test beds. Using AutoEval, creating a test bed is limited to writing a simple script describing the evaluation task.

Despite the large amount of existing evaluation software, we have not been able to find any previous reports on truly generic software systems for evaluation. The large amount of evaluation software further supports the need for a generic tool like AutoEval.

2.2 Features

AutoEval is a framework for automatic evaluation, written in C++. The main benefits of this generic evaluation system are the automatic handling of input and output and the script language that allows us to easily express complex evaluation tasks.

When evaluating an NLP system using AutoEval, the evaluation task is described in an XML configuration file. The configuration file defines what input files to be used and what format they are given in. Currently, AutoEval supports plain-text and XML files. The format of the data is defined using regular expressions. The system handles any number of input files.

The evaluation to be carried out is defined by a simple script language. Figure 1 provides an example of the script language. The three example lines are executed for each line of the input file. In line (1), the handle `file` identifies a row-based text file. The function `field` extracts tokens from the input file. The extracted tokens are assigned to variables that are later used for processing of the data. When the input file is depleted, the `field` function will fail and the data processing is completed. Line (2) increments a counter `tot` every time a new line is processed and will thus count the number of rows in the input file. Line (3) increments a counter variable that has the same name as the tag read. That is, if the tag read was e.g. *NN1*, the counter variable named `NN1` will be incremented by one. This provides an extreme flexibility as the script can create variables depending on the input. To further simplify variable processing and access, variables can be gathered in named groups. Normally, to access the string variable `var`, we write `:var`. To access data in a group `grp`, we would write `grp:var`.

The script language permits overloading of function names. That is, the same function name with different number of parameters will result in different function calls. If the basic set of functions is not sufficient for your task, you can easily add any C++ function of your own to the system. Thus, there is no limit to the expressiveness of the script language. Furthermore, common tasks (e.g. calculating precision and recall) that you use often can be collected in repository files where they can be accessed from all configuration files.

The output is written in XML. The user can choose to gather related output information under XML sections. To simplify output, the user can choose to output all variables in a given group. For example, in Figure 1, we can output all tag names and their corresponding frequencies without explicitly providing the PoS tag names, but only the group containing the variables (in this case the default group called `global`). The system can handle any number of output files.

AutoEval processes about 100 000 function calls (e.g. `field`) per second, or about 2000 rows (words) of input per second for the application in Section 4.

2.3 Usage

A short explanation of the use of AutoEval is given below. An example of the usage is given in Figure 1.

Prerequisites: (optional) a repository of functions to be used in the configuration script.

Input: a configuration file specifying the evaluation to be carried out. The input and output file names are provided in the configuration file.

Output: The results of the evaluation are given in XML. The data is divided into sections as specified by the configuration.

3 Missplel

Resources annotated with information on spelling and grammatical errors are rare and time-consuming to produce. Furthermore, it may be difficult to detect all errors and classify the errors found. Nevertheless, these resources are often useful when evaluating spelling checkers and grammar checking systems as well as other NLP system performance under the influence of erroneous or noisy input data.

Presumably, conventional corpus data is well proof read and scrutinized and thus, it is assumed not to contain errors. Given the corpus data, we will use an error introducing software called Missplel to introduce spelling and grammatical errors. This will provide us with the exact location and type of all errors in the file. This section reports on the features and implementation of Missplel.

3.1 Related Work

Several sources report on software used to introduce errors to existing text. Most of these deal mainly with performance errors or so-called Damerau-type errors, i.e. insertion, deletion or substitution of a letter or transposition of two letters (Damerau, 1964).

For example, Grudin (1981) has conducted a study of Damerau-type errors and from that, implemented an error generator. Agirre et al. (1998) briefly describe AntiSpell that simulates spelling errors of Damerau type. Another error introducing software, ErrGen, has been implemented in the TEMAA framework (Maegaard and others, 1997). ErrGen uses regular expressions at letter level to introduce errors, which allows the user to introduce Damerau-type errors as well as many competence errors, such as sound-alike errors (*receive, recieve*) and erroneously doubled consonants.

3.2 Features

The main objective in the development of Missplel was language and PoS tag set independency as well as maximum flexibility and configurability. To ensure language and PoS tag independence, the language is defined by a dictionary file containing word, PoS tag and lemma information. The character set and keyboard layout are defined by a separate file containing a *confusion matrix*, that is, a matrix holding the probability that one key is pressed instead of another.

Missplel introduces most types of spelling errors produced by human writers. It introduces performance errors and competence errors at both letter and word level by using four main modules: Damerau, SplitCompound, SoundError and SyntaxError. The modules can be enabled or disabled independently. For each module, we can specify an error probability. For example, if the Damerau module is set to a 10% probability of introducing an error, about 10% of the words in the text will be misspelled with Damerau-type spelling errors.

The Missplel configuration file, provided in XML, offers fine-grained control of the errors to be introduced. Most values in the configuration file will assume a default value if not provided. The format of all input and output files, including the dictionary file, is configurable by the user via settings using e.g. regular expressions.

Normally, misspelling *cat* to *car* would not be detected by a spelling or grammar checker. In Missplel, you can choose not to allow a word to be misspelled into an existing word or, if you allow existing words, choose only words that have a different PoS tag in the dictionary. This information (whether the error resulted in an existing word and if the tag changed or not) can be included in the output as shown in Figure 2.

```
Letters NN2
would   VM0
be      VBI
welcome AJ0-NN1

Litters NN2  damerau/wordexist-notagchange
would   VM0  ok
bee     NN1  sound/wordexist-tagchange
welcmoe ERR  damerau/nowordexist-tagchange
```

Figure 2: Missplel *example. The first part is the input consisting of row-based word/tag pairs. The second part is the* Missplel*ed output, where the third column describes the introduced error.*

The Damerau Module introduces performance errors due to keyboard mistypes (e.g. *wellcmoe*), often referred to as Damerau-type errors. The individual probabilities of insertion, deletion, substitution and transposition can be defined in the configuration and are by default equal between the four types. In the case of insertion and substitution, we need a probability of confusing one letter for another. This confusion matrix is provided in a separate file.

The Split Compound Module introduces erroneously split compounds. These errors are common in compounded languages like Swedish or German and may alter the semantics of the sentence. As an example in Swedish, *kycklinglever* (*chicken liver*) differs in meaning from *kyckling lever* (*chicken is alive*). A multitude of settings are available to control the properties (e.g. length and tag) of the first and second element of the split compound.

The Sound Error Module introduces errors the same way as ErrGen, that is, by using regular expressions at letter level. In Missplel, each rule has an individual probability of being invoked. This allows common spelling mistakes to be introduced more often. Using the regular expressions, many competence errors can easily be introduced (e.g. misspelling *bee* for *be*).

The Syntax Error Module introduces errors using regular expressions at both letter and word/tag level. For example, the user can form new words by modifying the tag of a word. This allows easy introduction of feature agreement errors (*he are*) and verb tense errors (*stop shout*). You can also change the word order, double words or remove words.

The foremost problems with resources annotated with errors are, for most languages, availability and the size of the resources. Using MISSPLEL, the only requirement is a resource annotated with word and PoS tag information, available for most languages. From this, we can create an unlimited number of texts with annotated and categorized errors.

MISSPLEL uses randomization when introducing errors into a text to be used for evaluation of the performance of an NLP system. To eliminate the influence of chance on the outcome of the evaluation, we may run the software repeatedly (say, n times) to obtain any number of erroneous texts from the same original text. The average performance on all texts will provide us with a reliable estimate on the real performance. The standard deviation will provide a clue to the *stability* or *robustness* of the system, i.e., the inherent ability of the NLP system to cope with various input. Low standard deviation would imply that the average is a good estimate on the real performance. Note here that the number of iterations n does not depend on the size of the annotated resource.

MISSPLEL processes about 1000 rows (words) of input per second for the application in Section 4.

3.3 Usage

A short explanation of the use of MISSPLEL is given below. An example of the usage is given in Figure 2.

Prerequisites: a row-based dictionary file, with each row containing a word, a PoS tag and a lemma.

Input: a configuration file specifying the error types to be introduced and a row-based data file with word and PoS tag data.

Output: a copy of the input file with errors introduced and an additional column that specifies the errors. Also, an XML file with more detailed information on the errors is produced.

3.4 Evaluation

An evaluation of MISSPLEL is being conducted to establish the quality of the produced errors. The main idea is to let human subjects decide whether a particular error is produced by MISSPLEL or a human. To this end, we used 15 rules for the SOUNDERROR module and 15 rules for the SYNTAXERROR module. The DAMERAU and SPLITCOMPOUND modules were also used.

The evaluation was divided into two parts. In the first part, the human was presented twenty independent sentences randomly chosen, each containing a highlighted error. The task was to decide whether the error was produced by a human or not. The second part consisted of two versions of the same text, both containing several errors. One of the versions contained errors introduced by MISSPLEL, the other contained genuine errors produced by the writer of the original text. The task was to decide which text contained the errors produced by a human.

To date, 42 human subjects have participated in the evaluation. In the first part, 62% of their guesses were correct. That is, in 62% of the sentences, the user was able to correctly guess that MISSPLEL produced the error or correctly guess that a human produced the error. This figure should be compared to the 50% level, where the user's guess is no better than a random guess. Thus, 62% is relatively close to the ideal result. In the second part, 52% (22 of 42) guessed correctly which text was produced by MISSPLEL. This is surprisingly close to the 50% random guess, which seems promising.

With more rules in the SOUNDERROR and SYNTAXERROR module, the ability to introduce human-like errors increases. Despite the limited amount of rules used, the results are promising. The most surprising result of this preliminary evaluation is that the second part appears more difficult than the first. Normally, context would give the user a better picture of the "author", i.e., MISSPLEL in half of the texts. We recognize that the number of participants in the evaluation is too low to draw any general conclusions and include the figures here as an indication of the performance of the error introducer.

4 Applications

Clearly, the field of application of AUTOEVAL is very broad. It has been used e.g. for the evaluation of robustness and performance of parsers and taggers (see below) as well as in a thorough investigation on ensemble and majority voting in PoS tagging (Sjöbergh, 2003).

4.1 Robustness of NLP Systems

Here, we present an evaluation methodology in which both MISSPLEL and AUTOEVAL are used. The details are described in (Bigert et al., 2003).

Unrestricted text will inevitably contain spelling and grammatical errors. Normally, PoS taggers and parsers are foremost designed to process correct text, and errors in the input to an NLP system will affect its behavior. If an NLP system can successfully cope with noisy and ill-formed input, we say that it is *robust*. Naturally, the performance will be affected by the errors. For example, when the context becomes increasingly noisy, the analysis becomes more difficult, even for a human. Thus, if we are given e.g. a parser, we want to determine the rate of degradation of the performance when faced with increasingly noisy text. Furthermore, we may also determine which tagger is the most robust in combination with the parser.

To this end, we use MISSPLEL to introduce different levels of errors. For example, if we introduce errors in 10% of the words, a robust system should obtain a performance loss of around 10%. In the mentioned paper, we investigated the performance degradation of a Swedish shallow parser (Knutsson et al., 2003). Using AUTOEVAL, we gathered detailed information on degradation of each phrase type (e.g. noun phrase, prepositional phrase) as well as tagger and parser overall performance. To eliminate the influence of chance, we applied MISSPLEL 10 times per error level. From the results, we used AUTOEVAL to calculate the average performance and the standard deviation. The AUTOEVAL script contained 7 lines of processing code (51 function calls).

5 Conclusions

We have presented two generic tools for automatic evaluation. The source code of the programs is freely available from the web site (Bigert, 2003). The first program, AUTOEVAL, is constructed to simplify the processing of data during an evaluation. The second program, MISSPLEL, is used to introduce errors into text. We constructed an evaluation in order to assess the quality of the produced errors from MISSPLEL. The results showed that the introduced errors are difficult to distinguish from authentic errors.

AUTOEVAL and MISSPLEL have successfully been used in several evaluation tasks. Based on our own experience of the two programs, we conclude that they greatly simplify the task of evaluation.

References

E. Agirre, K. Gojenola, K. Sarasola, and A. Voutilainen. 1998. Towards a single proposal in spelling correction. In *Proceedings of ACL 1998*, pages 22–28, San Francisco, California. Morgan Kaufmann Publishers.

J. Bigert, O. Knutsson, and J. Sjöbergh. 2003. Automatic evaluation of robustness and degradation in tagging and parsing. In *Proceedings of RANLP 2003*.

J. Bigert. 2003. The AutoEval and Missplel webpage. http://www.nada.kth.se/theory/humanlang/tools.html.

F. Damerau. 1964. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176.

J. Grudin. 1981. *The organization of serial order in typing*. Ph.D. thesis, Univ. of California, San Diego.

M. King et al. 1995. EAGLES – evaluation of natural language processing systems. http://issco-www.unige.ch/ewg95.

O. Knutsson, J. Bigert, and V. Kann. 2003. A robust shallow parser for Swedish. In *Proceedings of Nodalida 2003*, Reykjavik, Iceland.

B. Maegaard et al. 1997. TEMAA – a testbed study of evaluation methodologies: Authoring aids. http://cst.dk/projects/temaa/temaa.html.

K. Netter et al. 1998. DiET – diagnostic and evaluation tools for natural language applications. In *Proceedings of LREC 1998*, pages 573–579, Grenada.

M. Rajman et al. 1999. ELSE – evaluation in language and speech engineering. http://www.limsi.fr/TLP/ELSE/.

J. Sjöbergh. 2003. Combining PoS-taggers for improved accuracy on Swedish text. In *Proceedings of Nodalida 2003*, Reykjavik, Iceland.