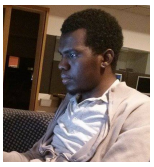
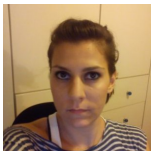
Supernova: Generating wardrobe tips depending on weather

Omar Elshenawy
09/10/1990
omares@kth.se

Mohamed Abdulziz
31/07/1985
moaah@kth.se



Thomai Stathopoulou
10/10/1987
thomai@kth.se



Varsha Kirani Gopinath
04/06/1991
varsha@kth.se

Abstract

Bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla

1 Introduction

Natural language generation (NLG) is an important subtask of many applications that rely on human computer interaction. While, as the Chinese proverb says, a picture is worth a thousand words, there are a lot of situations where pictures, schematic diagrams, maps, charts and plots cannot communicate information as effectively and efficiently as text would. There are no rules for when graphics is better than text. However, by means of observation, an annotated picture would better describe a physical location, charts can show progress, schematic diagrams can describe flow and hierarchy. All of them would fail, though, at describing abstract concepts such as causality or advice, which are quite straightforward to describe in text. Another issue on which the type of representation might depend is the expertise of the user, although graphical representation is considered an easy-to-understand way of representing information, research in psychology [1] has shown that in many cases, a considerable amount of information might be required to understand graphical representation and novice users might be better off with text.

In the current paper, an application that advices the user on what to wear with regard to the weather, is proposed. As noted earlier, text is probably the best way to communicate advices. A possible approach to this would be to simply provide a structured text, such as "Weather: Rainy, What to wear: Raincoat". Even though this might be informative, it is not user friendly. A better way to convey the message, could be naturally generated text with a friendlier tone, or a combination of text and graphics. However, due to limited time, resources and scope of the work, we attempt to generate plain natural language tips.

The system is built based on the structure of the Treebank-Trained Statistical Generator (TTSG) [2], which is explained and described in Section 2. Alternating and adding to this method, when possible, we attempt to take it apart, and examine the influence of every step of the procedure in the final result.

1.1 Contribution

As mentioned above, the system is built based on the structure of the TTSG method. Considering that this method, has been developed for a different application, the use of it for this particular "tip" application is an interesting endeavour. Moreover different approaches are attempted for different parts of the application and, finally, a new approach, based on a probabilistic model is proposed. This proposal is discussed in Section [reference].

1.2 Outline

In Section 2 we overview different attempts in field of natural language generation such as TreeBank & Context Free Grammar based architectures. We also briefly overview different common evaluation methods for NLG systems. In Section 3 we describe our implementation details, discussing different practical aspects such as design of corpus. In Section 4 we describe in detail our evaluation experiment for the different parameters that have been used and discuss the effectiveness of

the methodology of evaluation. Finally, in Section 5 we discuss the strengths and weaknesses of the variations of the method in light of our application.

2 Related work

In this section, certain methods and concepts will be discussed. Some of these methods and concepts, define the basic structure, upon which the application was implemented.

2.1 Probabilistic Context-Free Grammars (PCFGs)

2.1.1 An introduction to Context-Free Grammars (CFGs)

A Context-Free Grammar (CFG) is a 4-tuple $G = (N, \Sigma, R, S)$, where:

- N is a set of non-terminal symbols
- Σ is a set of terminal symbols
- R is a set of rules of the form $X \rightarrow Y_1 Y_2 ... Y_n$, where $X \in N, n \geq 0$ and $Y_i \in (N \cup \Sigma) \forall n = 1, 2, ...n$
- $S \in N$ is a distinguished start symbol[1]

A CFG creates sequences of symbols, replacing non-terminal symbols with a combination of non-terminal and terminal symbols, using the rules in R. The procedure of replacing symbols stops, when there are no non-terminal symbols left to be replaced. Every time a non-terminal symbol is replaced, the surrounding symbols are not taken into consideration, hence the term "Context-Free".

When creating a CFG for a language (e.g. English), usually non-terminal symbols represent basic syntactic and grammatical categories, whereas terminal symbols represent actual words. A small example of the rules of a CFG is shown in Table 1. It is obvious in this example, that the non-terminal symbols can either be generic syntactic terms (i.e. Noun Phrase, Verb Phrase etc.), or specific grammatical terms (i.e. Noun, Verb, Pronoun etc.).

Table 1: Example CFG

(a) General CFG rules

Sentence (S) → VerbPhrase NounPhrase
VerbPhrase → Verb VerbPhrase → Verb Preposition
NounPhrase → Pronoun Noun

(b) Unary rules

Verb → 'take' Verb → 'put'
Noun → 'raincoat' Noun → 'shoes'
Pronoun → 'your'
Preposition → 'on'

There is no restriction when it comes to the rules, as long as the symbol on the left-hand side of the rule is a *non-terminal*. The symbols on the right-hand side, can

[1]The definition of a CFG is borrowed from [3]

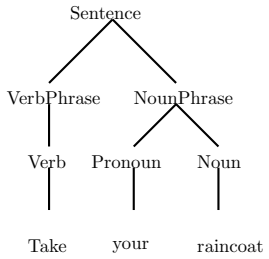


Figure 1: The parse tree of the sentence "Take your raincoat"

be any combination of symbols (terminal and/or non-terminal), or even an empty string (ϵ). A very specific kind of rule, is the unary rule. These rules have only *one* symbol on the right-hand side and this symbol is always a terminal symbol (Table 1b). The set of the unary rules can also be described as a lexicon, i.e. a tag for every word in the language (or a subset of the used language), representing what part of speech each word is (part-of-speech tagging, POS tagging).

2.1.2 Parse Trees

As mentioned in Section 2.1.1, given a CFG, one can produce a sequence of symbols (i.e. a sentence), simply by replacing non-terminal symbols with other symbols, according to the rules in R. One very common method is, starting with the start symbol S, to always replace the left most non-terminal symbol of the sequence, until there are only terminal symbols left. The set of rules that were used, in order to create the sentence, can be represented as a tree. This tree is called a **Parse Tree**. Figure 1 shows the parse tree of the sentence "Take your raincoat", based on the very simple example of the previous Section.

Because of the fact that there can be different rules with the same non-terminal symbol on the left-hand side (the same non-terminal symbol can be replaced by different combinations of non-terminal and terminal symbols), there can be more than one parse trees for one sentence. Additionally, given a sentence there a different methods (parsers) to generate parse trees. More details about different parsers are discussed in Subsection [reference subsection].

2.1.3 Probabilistic Context-Free Grammars (PCFGs)

Given a CFG, as defined in Section 2.1.1, a Probabilistic Context-Free Grammar (PCFG), can be defined as an extension of the CFG. Therefore, it consists of the 4-tuple $G = (N, \Sigma, R, S)$, as well as the new variable:

$$q(\alpha \to \beta) \forall rule \in R$$

The new variable represents the conditional probability of choosing rule $\alpha \to \beta$, given that the left-hand side non-terminal symbol is α. So, for any non-terminal symbol X, there is the constraint:

$$\sum_{X \to \beta \in R} q(X \to \beta) = 1 \text{ [3]}$$

Given a PCFG, one is able to calculate the probability of a parse tree, by multiplying the probabilities of all the rules that have been used for the production of the sentence.

In order to produce a PCFG based on a CFG, one would need to have a corpus, i.e. a set of sentences (symbol sequences) that can be produced by the given CFG. These sentences are converted into parse trees and the occurrences of every rule are counted. In the end the probability of every rule is the number of the rule's occurrences divided by the number of the occurrences of rules with the same non-terminal symbol on the left-hand side:

$$q(\alpha \to \beta) = \frac{Count(\alpha \to \beta)}{Count(\alpha)}$$

After having created a PCFG, the next step would to actually try and generate sentences based on that particular grammar. For that, exist different methods, that are to be be discussed in Section [reference].

2.2 Treebank-Trained Statistical Generator (TTSG)

The method described in [2] uses a CFG and a raw corpus. A raw corpus is nothing more, than a set of sentences written in natural language (e.g. English). Given that raw corpus and the CFG, the corpus is converted into a Treebank. That means that, for every sentence in the corpus, a parse tree in generated. The set of the different parse trees constitute the Treebank.

Using the generated Treebank, a PCFG is created as described in Section 2.1.3. Two different generation logics are followed at the last step of this method. The greedy and the Viterbi method are used and compared to one another, as well as to a random generator, i.e. at each step of the generation process, choose randomly a rule that applies, regardless of the rule's probability.

2.3 Evaluation Methods

Evaluating NLG systems is not a straightforward task. The quality of text can vary with regard to both the reader and the writer. Writers might not always produce text that is optimal from the reader's point of view, and text that is optimal from the reader's point of view, might not be optimally written. Also, since the target user of any NLG system is usually human, a variance in assessment will exist due to the variation in human understanding. Evaluation methods in the current literature can be grouped into three categories[7], each tries to assess the NLG system from different perspectives. In Section 4, we give details into our experiment and explain the methodology we used for evaluation.

Current evaluation methods can be grouped into three categories [7]:

- Human Rating and Judgement
- Automatic Metrics

2.3.1 Human Rating and Judgement

In this group of methods, humans are asked to rate and judge the text generated by the NLG. This evaluation can be done in several ways. One way is to provide users with text that is written by humans along with that of the NLG system, and asking them to rate both, without knowing which is written by which. Another is asking them to rate the quality of the text on a five-point scale, or asking them to choose their favourite text.

2.3.2 Automatic Metrics

Evaluation methods involving humans are usually quite expensive, time consuming, hard to design and hard to repeat. For these reasons, there has been a growing interest in evaluating NLG systems by comparing them to a reference-text corpus created by human experts using automated metrics such as string-edit distance, tree similarity and BLEU [6]. These methods don't require many resources and allow for repetitive immediate evaluation of the system. However, there exist some concerns for using these methods. Among them, is the fact that the reference-text corpora created by experts are optimal from a writer's point of view, which might not be optimal from a user's point of view, therefore the test is for similarity to writer's text.

3 My method

In the Section, the combination of the different concepts of the previous Section is going to be discussed. Apart from implementing the method described in 2.2, certain additions or variations are proposed and implemented.

3.1 Implementation

As is already mentioned, the basic structure of the generation system follows the structure of TTSG (2.2). Trying to describe the flow of the system, one can divide it into different parts, that are almost independent from each other:

- Classification of the input data
- Creation of CFG
- Creation of corpus & lexicon
- Creation of PCFG
- Tip generation

Each part is described in the following Sections.

3.1.1 Classification of the Input Data

The input data of the application, are going to be four variables. The variables will be four numbers concerning four different aspects of a weather forecast:

Table 2: Different weather tags and their corresponding thresholds

Temperature	**Cloud Coverage**	**Rain Precipitation**	**Wind Speed**
$\leq 15^oC \rightarrow$ Cold	$\leq 50\% \rightarrow$ Sunny	$\leq 50\% \rightarrow$ Dry	$\leq 11m/s \rightarrow$ Calm
$> 15^oC \rightarrow$ Hot	$> 50\% \rightarrow$ Cloudy	$> 50\% \rightarrow$ Rainy	$> 11m/s \rightarrow$ Windy

- Temperature: A number reflecting the temperature in degrees ^oC
- Cloud Coverage: A number [0 - 1] reflecting the percentage of cloud coverage
- Rain precipitation: A number [0 - 1] reflecting the probability of rain
- Wind Speed: A number reflecting the speed of the wind in m/s

After receiving these four parameters, they are being classified, by applying different thresholds (for the sake of simplicity, thresholds were chosen, instead of a more complex statistical model). This procedure, generates four tags, that constitute the weather forecast of the upcoming day. The different thresholds and their corresponding tags are shown in Table 2. For simplicity, only very distinct weather conditions were chosen.

3.1.2 Creation of CFG

The CFG, on which the tip generation is based, has been manually created, based on the basic rules in [3] and [5]. Some fundamental grammar and syntax rules of the English language are defined, making sure that they cover the entire available corpus and the needs of the application and, at the same time omitting the rules that were thought to be too detailed and complex.

3.1.3 Creation of corpus & lexicon

The corpus is also manually created. It is a set of fifty-one "wardrobe tips", along with tags for the corresponding weather tag of every tip.

The lexicon used is the Treebank Lexicon [4], with a few addition and modifications, to suit the needs of the application.

3.1.4 Creation of PCFG

The concept for the creation of the PCFG is as follows. After receiving the input data, and creating the different weather tags, the tips of the corpus are filtered based on their corresponding tags. The tips, whose tags, coincide with the weather conditions, constitute the corpus used, thus ensuring that the system is trained on the correct data and avoiding tips that are inappropriate to the current weather.

In order to create the PCFG, the corpus needs to be converted to a Treebank. At this point, different parsers (names?) are used for the creation of the Treebank, while testing the influence that they have in the final result.

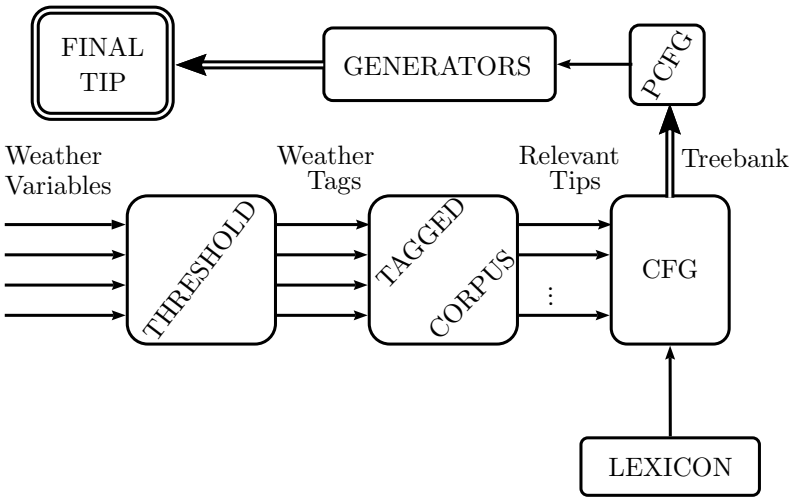


Figure 2: Flow Chart of the Implemented System

3.1.5 Tip generation

After the PCFG is created, different generators are used, in order to complete the final part of the process, which is the creation of the tip. The generators used are the greedy, Viterbi and random generator, just like [2], as well as the ProbGen generator, introduced in Section [reference].

Figure 2 shows a flowchart of the basic steps of the system that was implemented.

3.2 Parsers

The main task of parser is to transform natural language sentences to formal and symbolic representation in the form of trees where syntactic analyses is linked to each string (section parse tree?). These parse trees are collectively called as treebank. We use five different parsers, namely

- Chart Parser
- Earley Chart Parser
- Bottom Up Chart Parser
- Top Down Chart Parser
- Recursive Descent Parser(top down,depth first)

Chart parsing is an example of dynamic programming in the sense that we parse a sub part of input, we store the intermediate results and re-use them when appropriate. Intermediate results are stored in chart.As a result, chart parser doesn't do same work multiple times unlike top-down parser. This prevents both backtracking and combinatorial explosion.

Bottom Up Chart parsing starts from the input string and lookup CFG to find the relevant words or phrases in the right hand side of CFG. The parser replaces these with the left hand side of CFG until the whole sentence is reduced to S(root node).These intermediate results are recorded as edges on a chart. As name suggests, it is one of the variant of chart parser.

Top Down Chart parser starts off with a top level goal, which is from root node (S). This is broken down into sub-goals to find different constituents according to our CFG.These are expanded until a terminal is found. Once terminal is found, it is matched with input string.Accordingly it is added as edge to the chart.And the chart is completed once all the input strings are matched. Here chart is nothing but parse tree.

Earley chart parser is a variant of chart parser and it is based on top down parsing. But it deals more efficiently with matching against input strings by considering both top down and bottom up strategy.

Recursive descent parsing is as same as top down parsing but without using chart to store intermediate results.It executes same work multiple times.

All the five parsers generate similar parse trees in our application because of the naivety of our CFG.Due to this, we consider only chart parser. All the above mentioned parsers are implemented using NLTK(cite?)

3.3 Generators

Once we obtain treebank, we use them to develop PCFG(section PCFG). Thats how the system learns about corpus and we give this PCFG along with weather conditions as input to the generator. The generator will give out the sentence for that particular weather condition according to PCFG . We have implemented five generating algorithms:

ProbGen Generator In this method, the generator chooses between the rules in PCFG according to their probabilities randomly. For example: If PCFG has two rules A $->$ B and A $->$ C with probabilities 0.7 and 0.3, then ProbGen chooses rule (1) 7 times out of 10 and rule(2) 3 times out of ten.

Greedy Generator Greedy generator will choose the production with maximum probability at each node according to PCFG.

Viterbi Generator Unlike greedy generator, viterbi generator will choose the production with maximum probability along its path from root node(S)

Random generator Random generator chooses the production randomly irrespective of its probabilities.

n-gram Generator We have implemented n-gram generator with order 3 with words being each unit.(cite?)

4 Evaluation

The system initially filters through training sentences using the input. Therefore, checking for word relevance in the generated sentences to the input is not required. Also, any grammatical evaluation is skipped, because it is exclusively dependent on the initial CFG which if written carefully, would not allow for any grammatical mistakes in the generated output. For the evaluation, two types of tests were considered, automated similarity measures, namely, String-Edit distance and BLEU[6],

and a human perception test through an online survey. The methods' generalization performance is calculated through cross-validation using the automated metrics. However, String-edit and BLUE have a limitation that the semantics of the generated sentences are not considered. This gap was filled by the human perception test. The participants were presented with groups of sentences that are mix of human written and computer generated sentences and were asked to select the sentences, that they believed are written by a human, taking into consideration syntax, semantics and relevance to the input weather conditions.

4.1 String-Edit distance

The String-edit metric is based on the standard String-edit distance between two sentences, which is the sum of all additions, moves and substitutions needed to transform the first sentence into the other. For example the String-edit distance between "take your hat today" and "wear your hat" is two. The lower the distance, the more similar the two sentences are.

4.2 BLEU

BLEU is an evaluation metric for machine translation, and has recently been adopted by NLG community for its effectiveness. It is based on a weighted average of different n-grams as a measure of similarity. Every generated sentence is compared to a set of reference sentences and the score is calculated for every reference sentence. The minimum score is picked from the reference scores to account. First, it counts commonly occurring uni-grams (i.e. words) between both reference and generated sentences and divides by the total number of uni-grams in the generated sentence. It then computes the same count again for bi-grams, tri-grams, ..., n-gram occurrences and uses a weighted average sum of these counts. It is worth noting that these counts decay exponentially, therefore, BLEU uses the average logarithm with uniform weights. BLEU has shown correlation with monolingual human judgement using a maximum n-gram of order 4-grams. Therefore, that is what was used for the experiments that were conducted.

4.3 Experimental Setup

The built corpus was small; it contained 51 lines and 1438 words, of which 51 were unique. The corpus was divided into almost three equal folds to perform a three fold cross validation. The Treebanks were created using different parsers, such as chartParser, RecursiveDescentParser ADDD MORE PARSERS parsers. Then, the PCFG was created using the training fold of the corpus and, finally, for all combinations of the weather tags in the validation fold, a tip is produced by Greedy, Viterbi, randGen, n-gram and probGen generation methods. For the tips generated by all different parsers and generators, string edit distance and BLEU metric were calculated in regard to all the sentences in the validation folds. For the randGen and probGen methods, an average value over thirty repetitions is calculated. After

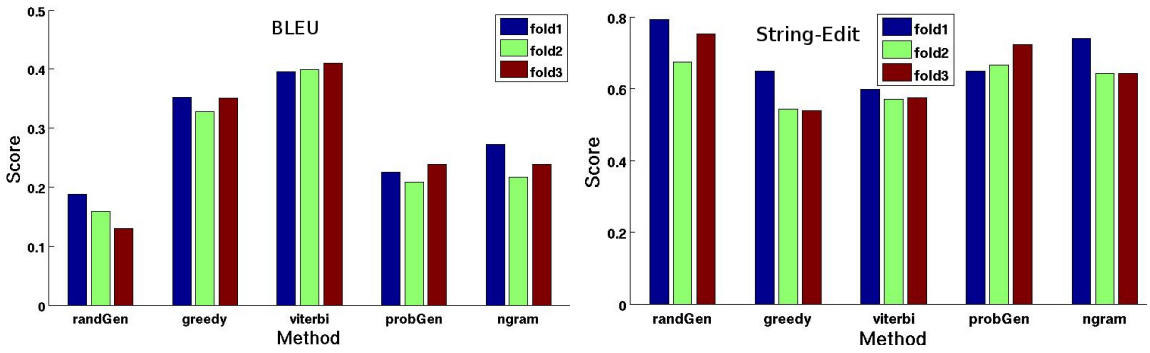


Figure 3: Model results for BLEU and String Edit

the metrics are calculated for every tip, the average value over all the sentences in each fold is computed.

4.4 Results

It has been observed that all parsers generated the exact same parse trees for the CFG. This is due to the simplicity of the CFG. Automated metric evaluation results for every method and every fold are presented in Figure 3. It is noticeable that, the performance of Viterbi is better than Greedy. This is consistent with [2]. However, unlike [2], the best algorithm in this evaluation was Viterbi, not n-gram. All the methods were better than randGen, however, probGen and n-gram had only a mild enhancement over randGen.

4.5 Discussion

The results we obtained from the automated metrics are measures of similarity between the generated sentences and some reference sentences. However, these metrics are unable to test if the generated sentence does in fact make sense with regard to the weather conditions. For instance, comparing sentence 1 and sentence 2 results in a large error for both metrics, however, both sentences are correct responses for a rainy weather. Representing this relationship between the sentence and weather condition can be done by making an extensive corpus, that would allow for many versions of the same sentence, such that it is guaranteed that sentences in a given fold, can be generated by sentences in the remaining folds. However, for a small corpus, an enhancement would be to account for the variation in the folds. That is, comparing the methods' performance for each testing fold, to the average similarity of the testing and training folds. This allows for a better yet still incomplete insight to the methods' performance. As seen in Table 3, the string-edit error of the best methods is slightly better than the fold to fold error. A corollary, is also that some sentences which had a low score on the automated metrics, had a good score on the human evaluation metrics.

1. Wear your raincoat
2. Take your umbrella

Table 3: Fold to fold error and comparison to Viterbi and Greedy

Fold	Fold Error	Viterbi Error	Greedy Error
Fold 1	0.528074795575	0.597222222222	0.647916666667
Fold 2	0.532187049062	0.569444444444	0.54375
Fold 3	0.53913270757	0.575	0.539583333333

5 Summary and Conclusions

bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla

References

[1] *Building Natural Language Generation Systems*, chapter 2 - National Language Generation in Practice. Cambridge University Press, 2000.

[2] Anja Belz. Statistical generation: Three methods compared and evaluated. 2005.

[3] Michael Collins. Probabilistic Context-Free Grammars (PCFGs). *Lecture Notes*, pages 1–18, 2013.

[4] Julia Hockenmaier and Mark Steedman. Acquiring compact lexicalized grammars from a cleaner treebank. In *Proc. 3rd International Conference on Language Resources and Evaluation*.

[5] James H Martin and Daniel Jurafsky. Context-Free Grammars for English. In *An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*, chapter 12.

[6] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*, page 311, Morristown, NJ, USA, 2001. Association for Computational Linguistics.

[7] Ehud Reiter and Anja Belz. An investigation into the validity of some metrics for automatically evaluating natural language generation systems. *Computational Linguistics*, (December 2008), 2009.