A sentence generator for Dutch

Daniël de Kok, Gertjan van Noord

University of Groningen

Abstract

The paper presents an efficient, wide-coverage, sentence generator for Dutch, which employs the Alpino grammar and lexicon. This generator consists of a chart-based sentence realizer that builds grammatical sentences for a given abstract dependency structure, and a maximum-entropy fluency ranker which selects the most fluent sentence from a set of candidate sentences for a given dependency structure. The coverage, speed and accuracy of the generator is evaluated on several corpora.

1 Introduction

Sentence realizers have been developed for various languages, including English and German. While the generation algorithms used in sentence realizers are very generic, the implementation of a realizer is quite specific to the grammar formalism and input representation. This paper describes a sentence realizer for the wide-coverage Alpino grammar and lexicon.

Alpino (van Noord 2006) is a parser for Dutch which includes an attribute-value grammar inspired by HPSG, a large lexicon, and a maximum entropy disambiguation component. Dependency structures are constructed by the grammar as the value of a dedicated attribute. These dependency structures constitute the output of the parser. Detailed documentation of the dependency structures is given in van Noord et al. (2010).

In generation, the grammar is used in the opposite direction: we start with a dependency structure, and use the grammar to construct one or more sentences which realize this dependency structure. In the general case, a given dependency structure can be realized by more than a single sentence. For instance, the sentence *Na de verkiezingen beklijfden de adviezen echter niet* (*After the elections the advises did, however, not persist.*) is mapped to a dependency structure which can also be realized by variants such as *Na de verkiezingen beklijfden de adviezen niet echter*, or *echter beklijfden na de verkiezingen de adviezen niet*. Therefore, a maximum entropy fluency ranker is part of the generator. The fluency ranker selects the most appropriate, 'fluent', sentence for a given dependency structure.

1.1 Dependency Structures

Various different abstract sentence representations have been proposed as input for sentence realization algorithms, such as Minimal Recursion Semantics (Copestake et al. 2005) and dependency structures (Hays 1964). A useful representation conforms to three characteristics: the representation should be easy to process by external applications; the representation should be abstract enough to map to interesting variation in realizations; and the representation should be native to the

grammar and lexicon.

Dependency structures can be argued to conform to these three characteristics. There is plenty previous work based on dependency structures, such as sentence compression (De Kok 2008), machine translation (Lin 2004), and sentence fusion (Marsi and Krahmer 2005), demonstrating its usability. Dependency structure is also native to the Alpino grammar and lexicon.

For the sentence realizer, we use the same dependency structure as created by the Alpino parser, except that we remove information about word order and word inflection. Words are represented in the dependency structure by their root forms, plus optional additional POS-tag information to select for a specific reading. The POS-tag information is presented by attributes such as *pos* with values *verb, noun,* ..., and *num* with values *sg,pl*. Underspecification of such attributes in the input is possible. For instance, leaving out the attribute for number (*num*) for a noun might realize the noun in singular or plural.

2 Sentence realization

2.1 Representation of dependency structures

The attribute-value grammar underlying Alpino constructs dependency structures by means of unification. Dependency structures are represented by attribute-value structures. A category such as *np* has a special attribute *dt* which represents its dependency structure.

The attribute-value structure representation includes information of the head word of that dependency structure, as well as attributes such as *su, obj1, obj2, mod, det* for each of its dependents (subject, direct object, secondary object, modifier, determiner). If there can be multiple dependents of the same type (e.g. for modifiers), a list-valued attribute is used. A special atomic value is used to represent the lack of a dependent of a particular type. For instance, if the input does not contain a direct object dependency, then the value of the attribute *obj1* will be the atom *nil*.

Consider, for example, the dependency structure in Figure 1(a). The word *adviezen* (*advises*, represented by the root *advies*) has one dependent, *de* (*the*) with the relation *det*. The word *beklijven* (*to persist*) is represented by the root *beklijf*, and takes the dependency structure associated with *advies* as its subject. This dependency structure is represented by the attribute-value structure shown in Figure 1(b). In this structure we only list the attributes which have a value different from the special value *nil*.

Sentences are realized using a bottom-up chart generator. The generator assumes that grammar rules contain the attribute-value structure of the mother node, and a list of attribute-value structures for each of the daughter nodes. Moreover, one of the daughters is identified as the head of the rule (typically a daughter is selected as the head if its dependency structure is equal to the dependency structure of the mother). The chart generator is described in Section 2.2.

In order that the generator constructs partial analyses that are realizing the input

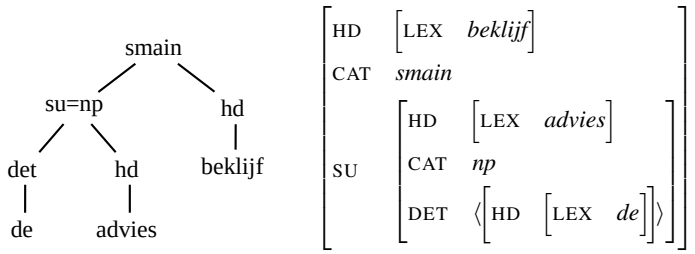

Figure 1: (a) Dependency tree and (b) attribute-value structure for *De adviezen beklijven.*

dependency structure, top-down guidance is crucial. Top-down guidance requires that every category considered during generation contains a dependency structure which unifies with a part of the input dependency structure. Top-down guidance is explained in more detail in Section 2.3.

During generation, partial realizations are packed in a realization forest for efficiency. Full realizations can be extracted from the packed representation, potentially using the fluency model described in Section 3 to extract only the best possible realization(s). Packing and unpacking are described in Section 2.4 and Section 2.5.

2.2 Chart generation

Chart generation (Shieber 1988, Kay 1996) closely resembles bottom-up chart parsing. A bottom-up chart parser builds derivations bottom-up from the words, chart generation also starts from the words. However, since the ordering of the words is yet to be determined, the worst-case time complexity of chart generation is exponential rather than cubic. This worst-case scenario is observed in practice when we consider modifiers: if a word has k modifiers, often 2^k orderings are admitted by the grammar. Still, chart generation is a relatively efficient algorithm, since partial analyses are constructed only once.

The algorithm schema of chart generation is simple: new edges are put on an agenda. During every iteration, the generator processes one edge from the agenda, and attempts to combine it with edges from the chart. This may lead to new edges. After an edge is processed, it is removed from the agenda and placed on the chart.

An edge represents a grammar rule that is partially or fully completed. Completion here means the process of filling daughter slots. We use two types of edges: *inactive edges* that have all daughters completed and *active edges* that have uncompleted, *active*, daughter slots. An active edge contains the attribute-value structure of the mother of a rule, and the list of attribute-value structures of the active daughters of a rule. An inactive edge only contains an attribute-value structure for the mother node.

During initialization, all relevant lexical entries are added as inactive edges to the agenda, as well as all mother categories of rules without any daughters (epsilon

rules).

When an active edge is processed, the chart is inspected for an inactive edge whose attribute-value structure unifies with the active daughter. Processing an inactive edge involves the same mechanism: find an active edge on the chart such that its active daughter unifies with the inactive edge. In addition, an inactive edge can be used to initialize an active edge if there is a grammar rule with a head daughter that unifies with the inactive edge.

If an edge on the agenda has been processed in all possible ways, it is removed from the agenda and put on the chart. Generation ends when the agenda is empty. Inactive edges with a dependency structure which equals the input dependency structure represent successful realizations.

2.3 Top-down guidance

When chart generation is finished, we are only interested in those derivations where the resulting dependency structure equals the input dependency structure. The input dependency structure constitutes our top-down information. During processing, we only need to construct edges which can be part of a derivation which leads to this input dependency structure.

Since the Alpino grammar is highly lexicalized, top-down information can be enforced in a very efficient manner. The dependency structure of the left hand-side of a rule is built up by unification on the basis of the dependency structures of each of the daughters of the right hand side, in such a way that all dependency information in lexical items ends up as a part of the dependency structure of the top node of the derivation tree. As a consequence, it is possible to 'inject' expected dependency information in the attribute-value structure of the lexical items.

Note that this mechanism is related to Shieber's semantic monotonicity requirement (Shieber 1988), but not identical. In our case, we do assume, as in (Shieber 1988), that the grammar exhibits the monotonicity requirement with respect to dependency structure. But we exploit this requirement one step further: the 'semantics' of a lexical entry is *instantiated* with part of the goal 'semantics'. As a consequence, our bottom-up algorithm is more goal-directed.

As an example, consider the dependency structure given earlier in Figure 1(a). During initialization of the chart, lexical look-up is performed with the additional requirement that the value of *dt* unifies with 1(b) or with a sub-part of 1(b). This implies that only the words *de, advies, adviezen* and inflectional variants of the verb *beklijven* are selected during lexical look-up. Moreover, the dependency structure of those lexical entries is already instantiated with parts of the dependency structure of 1(b). For instance, the lexical entry for *beklijfden* is given in Figure 2(a). Top-down guidance results in the entry given in figure 2(b). The example is simplified for expository purposes.

In Figure 2(a), the attribute-value structure states that the subcat list (the attribute SC) of the verb contains a single entry which is identical to the subject (attribute SUBJ). The dependency structure associated with the subject is identical to the SU of the dependency structure of the verb. In addition, the verb can take an

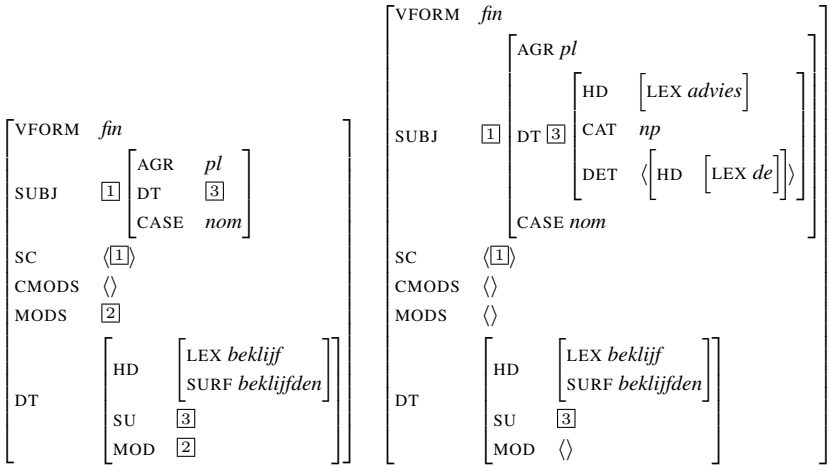

Figure 2: Attribute-value structure for *beklijfden* before (a) and after (b) top-down guidance.

arbitrary amount of modifiers. The pair of attribute CMODS and MODS are used to collect those modifiers in a derivation; CMODS contains the list of modifiers found so far, whereas MODS represents the complete list of modifiers.

In Figure 2(b), the dependency structure has been instantiated as a result of top-down guidance. Therefore, the verb can only combine with a subject which has a dependency structure associated with *de adviezen*. Moreover, the attribute value structure indicates that there cannot be a single modifier attached to the verb, nor any other dependents.

Top-down guidance not only prevents too many dependencies to be constructed, but it also enforces that all required dependencies are found. For instance, if the input contains a dependency structure of a verb with a subject and a direct object, then lexical look-up will typically not propose an attribute-value structure for the intransitive reading of that verb: that attribute-value structure will have, in the lexicon, the value *nil* for the attribute *obj1* which will not unify with the goal. Similarly, if the input contains a number of modifiers associated with a head, then, typically, maximal projections of that head with fewer modifiers are also ruled out, solving one of the problems raised in (Kay 1996).

Top-down guidance is thus achieved by instantiating each lexical entry with part of the dependency structure of the input. In addition, Shieber's semantic monotonicity requirement is enforced for other edges as well. A new edge is constructed only in case its dependency structure is unifiable with part of the dependency structure of the goal.

2.4 Packing

For input with a high complexity the generation chart can grow enormously, while there may be many edges with the same attribute-value structure. For instance, the use of optional punctuation does not change an attribute-value structure, while it does introduce new edges for all allowed combinations of punctuation. Another source of growth of the chart are lexical items that have more than one valid inflection.

To compress the chart, we apply *packing*. In packing one inactive edge can represent multiple derivation histories with the same attribute-value structure. The history is represented by simple items, where each item is numbered by the inactive edge representing the item, and contains the identifier of the rule or lexical item used to construct the edge. In the case of a rule we also include a list of pointers to inactive edges that were used to fill the daughter slots of the rule.

For instance, the derivation history `his(31,r(top_start,[23,30]))` represents one particular way the attribute-value structure of inactive edge *31* was constructed, by completion of the *top_start* grammar rule using inactive edges *23* and *30*. Derivation histories representing terminal nodes contain lexical information, such as the root and the Alpino part of speech tag, rather than a list of inactive edge pointers.

Packing is performed when an inactive edge is completed. If an inactive edge is found on the chart with the same attribute-value structure, the history of the new inactive edge is added to the existing inactive edge. We have also experimented with forward-packing (where packing is applied when the new edge is subsumed by an inactive edge on the chart) and backward-packing (where the new edge subsumes an inactive edge on the chart). However, the benefits of both forms of packing did not outweigh the decreased performance caused by subsumption checking.

2.5 Unpacking

After chart generation, full and partial realizations can be retrieved (unpacked) from the packed forest. During unpacking, a derivation tree is created for these derivations. A full realization is represented by an edge with a top category and has a dependency structure that unifies with the target dependency structure. Derivation trees are constructed by expanding histories top-down. The algorithm for expanding a history recursively is shown in the following Prolog fragment:

```
unpack(Id,AttrVal,Tree) :-
    his(Id,His), unpack_his(His,AttrVal,Tree).

unpack_his(r(RuleId,Ds),LHS,tree(LHS,RuleId,Trees)) :-
    grammar_rule(RuleId,LHS,RHS),
    unpack_ds(Ds,RHS,Trees).
unpack_his(l(Lex),AttrVal,Tree,tree(AttrVal,Lex,[])) :-
    lexical_entry(Lex,AttrVal).

unpack_ds([],[],[]).
```

```
unpack_ds([Id|IdT],[AttrVal|AttrValT],[Tree|TreeT]) :-
    unpack(Id,AttrVal,Tree), unpack_ds(IdT,AttrValT,TreeT).
```

The `unpack` predicate retrieves a history with a particular identifier. The auxiliary `unpack_his` predicate performs the actual unpacking. If the history represents a non-terminal, the `unpack_his` predicate applies unpacking to all daughter identifiers. The attribute-value structure of the inactive edge is then reconstructed by retrieving the grammar rule, and completing the rule using the unpacked daughters. If the history represents a lexical node, we retrieve the attribute-value structure for this lexical node, and form a derivation tree leaf node.

Since multiple realizations can generally be generated for a dependency structure, there can be many realizations in the packed forest. Depending on the application, we may want to unpack all or a specified (N) number of realizations. We will use *N-best* unpacking to unpack a specific number of realizations, using a beam that retains only the most fluent realizations for histories representing maximal projections.

3 Fluency ranking

3.1 Introduction

Often many different realizations can be generated for a given input. While all realizations are grammatical, if the grammar is not too permissive, they often differ greatly in fluency. For this reason, we constructed a fluency ranker to select the most fluent realization.

3.2 Model

Different statistical models for fluency ranking have been proposed in the past, such as n-gram language models, maximum entropy models, and support vector machines (Velldal 2008). N-gram language models calculate the probability of a realization purely based on words, while maximum entropy models and support vector machines are linear classifiers that can integrate arbitrary features. Since feature-based models can integrate more information, they perform better than n-gram language models. As Velldal (2008) shows, maximum entropy models perform comparably to support vector machines for fluency ranking, while having a shorter training time. For this reason we use a maximum entropy model in our fluency ranker.

The principle of maximum entropy models is to minimize assumptions, while constraining the expected feature value to be equal to the feature value observed in the training data. In its canonical form, the probability of an event (y) within a context (x) is a log-linear combination of features (f_i) and their weights (λ_i) (Berger et al. 1996), normalized over all events in that context ($Z(x)$):

$$p(y|x) = \frac{1}{Z(x)} exp \sum_{i=1}^{n} \lambda_i f_i \quad (1)$$

The training process estimates optimal feature weights, given the constraints and the principle of maximum entropy. In fluency ranking, a dependency structure is a context, and a realization of that dependency structure is an event within that context.

In fluency ranking, we are not interested in the probabilities of the realizations of a given input, but in the ordering imposed by the probabilities. Since the normalization is constant for every realization given an input, we can simply calculate the linear combination of features and their values to assign a score to a realization:

$$score(y) = \sum_{i=1}^{n} \lambda_i f_i \tag{2}$$

3.3 Features

Since a maximum entropy model ranks realizations based on feature values, we have to settle on a set of features that can adequately describe characteristics of fluent sentences. Features for fluency ranking can be divided in two classes: (1) *output features* that describe aspects of the produced sentence, such as the frequency of a word n-gram in the sentence, and (2) *Construction features* that describe aspects of the process that constructed the sentence, such as the frequency of a particular rule being used to derive the sentence. Our current research does not integrate extra-sentential features.

Features can be hand-crafted or created by applying feature templates to training or evaluation data. A feature template can be seen as a function that has a derivation as its input and a set of features as its output. In the following two sections we describe the output and construction features that we use in our fluency ranker.

3.3.1 Output features

Currently, two output features are used, that represent auxiliary distributions (Johnson and Riezler 2000): trigram models for words and part-of-speech tags. For example, consider the sentence *de optische astronomie maakt gebruik van zichtbaar licht* (*the optical astronomy makes use of visible light*), and the corresponding sequence of part of speech tags (*determiner(de)..noun(het,sg,[])*).

We can use the word trigram model to estimate $P_{word}(de..licht)$ and the tag trigram model to estimate $P_{tag}(determiner(de)..noun(het, sg, []))$. The logarithms of these probabilities then become the values of the *ngram_word* and *ngram_tag* features, respectively. As shown in Table 1, these values are multiplied by the weights of the features that were found during training of the maximum entropy model. If we have no other features, the score of this realization is then the sum of the weighted scores.

Both models are trained on newspaper articles from the Twente Nieuws Corpus[1], consisting of 110 million words. For the part-of-speech tag trigram model

[1] `http://wwwhome.cs.utwente.nl/druid/TwNC/TwNC-main.html`

Feature	Weight (λ_i)	Value (f_i)	$\lambda_i \cdot f_i$
ngram_word	0.0158	-62.70	-0.9907
ngram_tag	0.0115	-24.05	-0.2766
Score ($\sum_{i=1}^{n} \lambda_i f_i$)			-1.2673

Table 1: Example output feature values for the sentence *de optische astronomie maakt gebruik van zichtbaar licht* (*the optical astronomy makes use of visible light*). Each value multiplied by the feature weight that was found during training of the maximum entropy model. The score of the realization is obtained by summing the weighted feature values.

we use the Alpino part of speech tags.

The probability of unseen trigrams are estimated using linear interpolation smoothing (Brants 2000), where unknown word probabilities are estimated with Laplacian smoothing.

3.3.2 Construction features

We experimented with construction features originating from parse disambiguation, as well as features specifically crafted for fluency ranking. The parse disambiguation features are used in the Alpino parser, and model linguistic phenomena that indicate preferred readings. Phenomena that are modeled include: topicalization of (non-)NPs and subjects; the use of long-distance/local dependencies; orderings in the middle field; identifiers of grammar rules used to build the derivation tree; and parent-daughter combinations.

Furthermore, we use some of features that were devised by Velldal (2008) for fluency ranking. These features describe local derivation sub-trees with optional grandparenting, including variants that contain the binned frequency and standard deviation of the words a sub-tree dominates over.

3.4 Feature selection

Since most features are extracted automatically using feature templates, many features are not necessary in an effective model because they occur only sporadically, correlate strongly with other features in the model (they show the same behavior within specific events), or have little or no correlation to the ranking.

Feature selection tries to extract $S \subset F$ from a set of features F, such that a model using S performs comparable to a model using F. This is particularly useful if $|S| \ll |F|$, which is true in our experiments where we compress the most effective model from 500,911 features to 2,200 features.

Frequency-based selection has often been used in the literature. In this method features that change often within the same context are selected. However, this approach does not account for overlapping or noisy features (De Kok 2010). For this reason, we use a maximum entropy selection method. In this selection method, a maximum entropy model is built one feature at a time, always selecting the feature

the brings the model closest to the training data. To make this computationally tractable, the method assumes that the weights of features that were previously selected are not affected by the addition of a new feature. Since noisy and overlapping features do not contribute to the model, they are not selected.

4 Evaluation methodology and results

4.1 Sentence realizer

We evaluated the sentence realizer using the Alpino test suites and sentences from Wikipedia. These test suites contain sentences, and their manually corrected dependency structures. The test suites are used to detect regressions in the parser, and are now used for the same purpose in generation.

To test the sentence realizer we keep track of the fraction of dependency structures for which realizations could be constructed (coverage), the number of realizations, and the time required for constructing all realizations for a given dependency structure. Using this information, we can extract various interesting characteristics, such as the coverage of the realizer, and the average generation time for a dependency tree of a certain complexity.

We have evaluated the sentence realizer using three Alpino test suites (the so-called *g*, *h*, and *i* suites). These suites where created during grammar development, are generally of increasing complexity, and cover a wide array of lexical and grammatical phenomena. Additionally, we have created a new suite for the evaluation of the sentence realizer, based on sentences of 5 to 25 tokens that were randomly selected from the Dutch Wikipedia of August 2008[2]. This suite was added to evaluate the sentence realizer and fluency ranker on real-world data. Table 2 shows the coverage of the sentence realizer for these suites.

Suite	**Inputs**	**$\geq$ 1 realization**	**Coverage (%)**
g_suite	996	996	100.0
h_suite	991	965	97.4
i_suite	271	262	96.7
Dutch Wikipedia	15398	13701	89.0

Table 2: Coverage of the chart generator on various test suites.

As we can see in this table, coverage is complete for the *g* suite, and very good for the *h* and *i* suites. Most of the problematic dependency structures from the *h* and *i* suites contain lexical information that could not be found in the lexicon, nor be handled through the productive lexicon. There are also some less interesting cases, such as empty dependency structures, derived from sentences that only contain punctuation. As expected, the coverage on the Wikipedia suite is lower. In this suite, the generator encounters more unknown roots and noise, such as equations and English phrases.

[2] `http://ilps.science.uva.nl/WikiXML/`

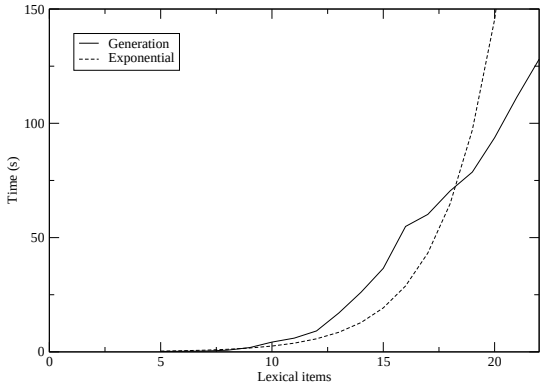

Figure 3: Average time for generating all realizations of dependency structures with a varying number of lexical items.

As described in Section 2.2, the worst-case time complexity of chart generation is exponential. Figure 3 shows the average time for generating all realizations of dependency structures with 5-22 roots, derived from the Wikipedia suite. The figure also shows the exponential after fitting with the average times. We can see that the amount of time required for generation grows enormously with the input complexity, but not yet exponentially.

4.2 Fluency ranking

4.2.1 Generation of data

The training and evaluation data for the fluency ranker was created by parsing the Wikipedia sentences described in Section 4.1 using the Alpino parser. The dependency structure corresponding to the best parse as selected by the disambiguation component was extracted and assumed to be the correct parse. Alpino achieves a concept accuracy of around 90% on common Dutch Corpora (Van Noord 2007). Furthermore, we assume that the sentence from Wikipedia is the most fluent realization of the extracted dependency structure. This assumption often does not hold in the strict sense. If we are charitable, we can assume that a writer expresses meaning in the most fluent manner, however this does not exclude the possibility that there are multiple fluent realizations.

We then use the chart generator to build the sentences that realize each dependency structure. The derivation trees and associated feature structures for these realizations are compressed, and stored in a derivation tree-bank. We also assign a quality score to each realization by comparing it to the original Wikipedia sentence

using the General Text Matcher (GTM) method (Melamed et al. 2003). Finally, we extract the fluency features of all trees in the derivation tree-bank.

4.3 Training of the model

To assess the usefulness of various classes of features, we train fluency ranking models with different sets of features. Each model is trained using 6786 training instances, where each training instance represents a dependency structure, and consists of the quality score and extracted feature values for each realization of that dependency structure. For each training instance, we randomly select 100 realizations to get a representative sample of the training data. A maximum entropy model is trained with a Gaussian prior of 0.001.

4.3.1 Quantitative evaluation

The fluency ranking models are evaluated by applying the rankers to 6785 test cases, counting how often each model chooses the most fluent realization from the set of realizations for a particular dependency structure. The realization with the highest GTM score is considered to be the most fluent realization in the set (*best match*).

Table 3 shows an example of this methodology. The table contains the realizations of a dependency structure obtained by parsing the phrase *de geletterdheid van de volledige bevolking wordt geschat op 36%* (*the literacy of the full population is estimated to be 36%*), along with their quality scores, and the fluency scores assigned by the fluency ranking model. If the fluency ranker assigns the highest score to the realization with the highest quality, it picked the most fluent sentence. In our evaluation, we only consider dependency structures with ≥ 5 realizations (5032 instances) to make the task difficult enough.

Realization	**Quality (GTM)**	**Ranker score**
de geletterdheid van de volledige bevolking wordt geschat op 36%	**1.00**	**-1.20**
de geletterdheid van de volledige bevolking wordt op 36% geschat	0.74	-1.56
op 36% wordt de geletterdheid van de volledige bevolking geschat	0.65	-1.66
op 36% wordt de geletterdheid geschat van de volledige bevolking	0.51	-1.81

Table 3: A fluency evaluation example for a dependency structure with four realizations. If the fluency ranker assigns the highest score to the realization that has the highest quality score (as in this example), the ranker has picked the most fluent realization.

For the evaluation of the accuracy of fluency ranking models we apply very mild feature selection using a low frequency cut-off, including features that change

their value within at least 4 contexts. Such a conservative selection has a negligible impact on the accuracy of the ranker, while making evaluation faster.

We have evaluated various different models consisting of different classes of features. In Table 4 we show the best match and GTM scores of the models that we evaluated. The first model (*ngram*) focuses purely on output features, integrating auxiliary distributions of the word and tag trigram models. Since this model is also trained using maximum entropy modeling, weights are assigned to each auxiliary distribution.

Model	**Best match (%)**	**GTM**
ngram	33.35	0.6266
ngram + parse	41.41	0.6586
ngram + velldal	43.60	0.6580
all	44.69	0.6633

Table 4: Best match accuracies and GTM scores for fluency models incorporating n-gram, parse disambiguation, and Velldal's generation features. The model using a combination of these features outperforms models that do no include one (or more) of these feature sets.

Adding parse disambiguation features (*ngram + parse*) or Velldal's generation features (*ngram + velldal*) to the model improves performance considerably. Here, Velldal's features seem to have more effect, although many of the features extracted using Velldal's templates indirectly describe the same characteristics as the more linguistically-motivated parse disambiguation features. Finally, combining all these feature classes gives the best model.

The best match accuracies in Table 4 may seem relatively low, however, these scores should be seen as an indication of the relative performance of each model. Since we created the training and testing data from Wikipedia sentences, there was no manual verification that the dependency structure used was the correct reading of the original sentence. Additionally, we only have the original sentence as an annotation, while there is often more than one fluent sentence. For this reason, the next section describes a qualitative evaluation.

4.3.2 Qualitative evaluation

We have performed a preliminary qualitative evaluation, by manually judging the realization that was selected by the fluency ranker with all features for 100 evaluation instances ourselves. To each selected realization, we assign one of three categories: fluent; neutral (fluent with some minor deficiency, such as missing punctuation or an harmless incorrect inflection); or non-fluent. We found that 80% of the evaluations was fluent, 8% neutral, and 12% not-fluent.

Nearly all of the realizations in the 'neutral' category had a missing comma where it would improve fluency, or an incorrect noun inflection. For efficiency, we currently generate with the minimum amount of punctuation possible. Allowing

more punctuation could improve readability. The incorrect noun inflections were produced by the productive lexicon.

Realizations in the 'not-fluent' category are more diverse, but a problem that we often encounter is ordering in coordinations. For instance, the fluency ranker currently has no preference for *de man en zijn hond* (*the man and his dog* over *zijn hond en de man* (*his dog and the man*), unless a particular case was seen in the training data.

4.3.3 Feature selection

Finally, we compress the best-performing model using feature selection. To compress the model, we first extract the best 10,000 features according to maximum entropy feature selection. We then create models of 100 to 10,000 features with a step size of 100 features. After plotting the accuracies of all models, we pick the amount of features that gives the same accuracy as the baseline model. The baseline model uses mild feature selection (excluding features with values that change within less than 4 contexts).

Method	Features	Accuracy (%)
no selection	500,911	44.83
baseline (cutoff-4)	90,521	44.69
frequency	8,900	44.12
maximum entropy	2,200	44.12

Table 5: Accuracies and model sizes after applying feature selection. Maximum entropy selection compresses the model enormously, while giving a accuracy comparable to other models.

As we can see in Table 5, only 2,200 features are required to make a model that performs comparably to a model with a fixed frequency-based cut-off. This model is tiny in comparison, and can give good insights in what features are important for fluency ranking (De Kok 2010). We also show the results of a frequency-based selection that picks the N most frequently changing features. This method results in a model that is more than four times larger than applying maximum entropy feature selection.

5 Conclusion

In this paper we have presented a sentence generator that leverages the Alpino grammar and lexicon to provide high-coverage realization from dependency structures. The sentence realizer uses a novel approach to top-down guidance, where information about expected and absent dependency relations is added to the dependency structure of lexical items. This ensures that only the required dependencies are present in derivations. The sentence generator also contains a fluency ranker which attempts to select the most fluent realization for a dependency structure.

In the future, we hope to improve the performance of the generator on very complex dependency structures. For instance, we are currently refining and experimenting with N-best extraction, for applications where only the most fluent realization is required. Additionally, we hope to add more linguistically inspired fluency ranking features to replace the Velldal (2008) features.

References

Berger, A.L., V.J.D. Pietra, and S.A.D. Pietra (1996), A maximum entropy approach to natural language processing, *Computational linguistics* **22** (1), pp. 71, MIT Press.

Brants, T. (2000), TnT – a statistical part-of-speech tagger, *Proceedings of the Sixth Applied Natural Language Processing (ANLP-2000)*, Seattle, WA.

Copestake, A., D. Flickinger, C. Pollard, and I.A. Sag (2005), Minimal recursion semantics: An introduction, *Research on Language & Computation* **3** (4), pp. 281–332, Springer.

De Kok, D. (2008), *Headline generation for Dutch newspaper articles through transformation-based learning*, Master's thesis, University of Groningen.

De Kok, D. (2010), Feature selection for fluency ranking, accepted (2010).

Hays, D.G. (1964), Dependency theory: A formalism and some observations, *Language* **40** (4), pp. 511–525, Linguistic Society of America.

Johnson, M. and S. Riezler (2000), Exploiting auxiliary distributions in stochastic unification-based grammars, *Proceedings of the 1st NAACL conference*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 154–161.

Kay, M. (1996), Chart generation, *Proceedings of the 34th annual meeting on ACL*, ACL, pp. 200–204.

Lin, D. (2004), A path-based transfer model for machine translation, *Proceedings of the 20th international conference on Computational Linguistics*, ACL, p. 625.

Marsi, E. and E. Krahmer (2005), Explorations in sentence fusion, *Proceedings of the European Workshop on Natural Language Generation*, pp. 8–10.

Melamed, D., R. Green, and J. Turian (2003), Precision and recall of machine translation, *HLT-NAACL*.

Shieber, S. (1988), A uniform architecture for parsing and generation, *Proceedings of the 12th COLING conference*, Budapest.

van Noord, G. (2006), **A**t **L**ast **P**arsing **I**s **N**ow **O**perational, *TALN 2006 Verbum Ex Machina, Actes De La 13e Conference sur Le Traitement Automatique des Langues naturelles*, Leuven, pp. 20–42.

Van Noord, G. (2007), Using self-trained bilexical preferences to improve disambiguation accuracy, *Proceedings of the 10th International Conference on Parsing Technologies*, ACL, pp. 1–10.

van Noord, G., I. Schuurman, and G. Bouma (2010), Lassy syntactische annotatie, revision 17566.

Velldal, E. (2008), *Empirical Realization Ranking*, PhD thesis, University of Oslo, Department of Informatics.