ML-Tuned Constraint Grammars

Eckhard Bick
Institute of Language and Communication
University of Southern Denmark, Odense
eckhard.bick@mail.dk

Abstract

In this paper we present a new method for machine learning-based optimization of linguist-written Constraint Grammars. The effect of rule ordering/sorting, grammar-sectioning and systematic rule changes is discussed and quantitatively evaluated. The F-score improvement was 0.41 percentage points for a mature (Danish) tagging grammar, and 1.36 percentage points for a half-size grammar, translating into a 7-15% error reduction relative to the performance of the untuned grammars.

1 Introduction

Constraint Grammar (CG) is a rule-based paradigm for Natural Language Parsing (NLP), first introduced by Karlsson et al. (1995). Part-of-speech tagging and syntactic parses are achieved by adding, removing, selecting or substituting form and function tags on tokens in running text. Rules express linguistic contextual constraints and are written by hand and applied sequentially and iteratively, ordered in batches of increasing heuristicity and incrementally reducing ambiguity from morphologically analyzed input by removing (or changing) readings from so-called readings cohorts (consisting of all possible readings for a given token), - optimally until only one (correct) reading remains for each token. The method draws robustness from the fact that it is reductionist rather than generative - even unforeseen or erroneous input can be parsed by letting the last reading survive even if there are rules that would have removed it in a different context. Typical CG rules consist of an operator (e.g. REMOVE, SELECT), a target and one or more contextual constraints that may be linked to each other:

(a) REMOVE VFIN (-1C ART OR DET) ;

(b) SELECT VFIN (-1 PERS/NOM) (NOT *1 VFIN)

Rule (a), for instance, removes a target finite verb reading (VFIN) if there is an unambiguous (C) article or determiner 1 position to the left (-), while rule (b) selects a finite verb reading, if there is a personal pronoun in the nominative immediately to the left, and no (NOT) other finite verb is found anywhere to the right (*1).

Mature Constraint Grammars can achieve very high accuracy, but contain thousands of rules and are expensive to build from scratch, traditionally requiring extensive lexica and years of expert labor. Since grammars are not data-driven in the statistical sense of the word, domain adaptation, for instance for speech (Bick 2011) or historical texts (Bick 2005), is traditionally achieved by extending an existing general grammar for the language in question, and by using specialized lexica or two-level text normalization. However, due to its innate complexity, the general underlying grammar *as a whole* has properties that do not easily lend themselves to manual modification. Changes and extensions will usually be made at the level of individual rules, not rule interactions or rule regrouping. Thus, with thousands of interacting rules, it is difficult for a human grammarian to exactly predict the effect of rule placement, i.e. if a rule is run earlier or later in the sequence. In particular, rules with so-called C-conditions (asking for unambiguous context), may profit from another, earlier rule acting on the context tokens involved in the C-condition. Feed-back from corpus runs will pinpoint rules that make errors, and even allow to trace the effect on other rules applied later on the same sentence, but such debugging is cumbersome and will not provide information on missed-out positive, rather than negative, rule interaction. The question is therefore, whether a hand-corrected gold corpus and machine-learning techniques could be used to improve

27th Pacific Asia Conference on Language, Information, and Computation pages 440－449

performance by data-driven rule ordering or rule adaptation, applied to existing, manual grammars. The method would not only allow to optimize general-purpose grammars, but also to adapt a grammar in the face of domain variation without actually changing or adding any rules manually. Of course the technique will only work if a compatible gold-annotation corpus exists for the target domain, but even creating manually-revised training data from scratch for the task at hand, may be warranted if it then allows using an existing unmaintained or "black box" grammar. Other areas where ML rule tuning of existing grammars may be of use, is cross-language porting of grammars between closely related languages, and so-called bare-bones Constraint Grammars (Bick 2012), where grammars have to cope with heuristically analyzed input and correspondingly skewed ambiguity patterns. In such grammars, linguistic intuition may not adequately reflect input-specific disambiguation needs, and profit from data-driven tuning.

2 Prior research

To date, little work on CG rule tuning has been published. A notable exception is the µ-TBL system proposed in (Lager 1999), a transformation-based learner working with 4 different rule operators, and supporting not only traditional Brill-taggers but also Constraint Grammars. The system could be seeded with simple CG rule templates with conditions on numbered context positions, but for complexity reasons it did not support more advanced CG rules with unbounded, sentence-wide contexts, barrier conditions or linked contexts, all of which are common in hand-written Constraint Grammars. Therefore, while capable of building automatic grammars from rule templates and modeling them on a gold corpus, the system was not applicable to existing, linguist-designed CGs.

That automatic rule tuning can capture systematic differences between data sets, was shown by Rögnvaldsson (2002), who compared English and Icelandic µ-TBL grammars seeded with the same templates, finding that the system prioritized right context and longer-distance context templates more for English than Icelandic. For hand-written grammars, rather than template expression, a similar tuning effect can be expected by prioritizing/deprioritizing certain rule or context types by moving them to higher or lower rule sections, respectively, or by inactivating certain rules entirely.

Lindberg & Eineborg (1998) conducted a performance evaluation with a CG-learning Progol system on Swedish data from the Stockholm-Umeå corpus. With 7000 induced REMOVE rules, their system achieved a recall of 98%. An F-Score was not given, but since residual ambiguity was 1.13 readings per word (i.e. a precision of 98/113=86.7%), it can be estimated at 92%. Also, the lexicon was built from the corpus, so performance can be expected to be lower on lexically independent data.

Though all three of the above reports show that machine learning can be applied to CG-style grammars, none of them addresses the tuning of human-written, complete grammars rather than lists of rule templates[1]. In this paper, we will argue that the latter is possible, too, and that it can lead to better results than both automatic and human grammars seen in isolation.

3 Grammar Tuning Experiments

As target grammar for our experiments we chose the morphological disambiguation module of the Danish DanGram[2] system and the CG3 Constraint Grammar compiler[3]. For most languages, manually revised CG corpora are small and used only for internal development purposes, but because Constraint Grammar was used in the construction of the 400.000 word Danish Arboretum treebank (Bick 2003), part of the data (70.800 tokens) was still accessible in CG-input format and could be aligned to the finished treebank, making it possible to automatically mark the correct reading lines in the input cohorts. Of course the current DanGram system has evolved and is quite different from the one used 10 years ago to help with treebank construction, a circumstance

[1] One author, Padró (1996), using CG-reminiscent constraints made up of close PoS contexts, envisioned a combination of automatically learned and linguistically learned rules for his relaxation labelling algorithm, but did not report any actual work on human-built grammars.

[2] An description of the system, and an online interface can be found at: http://beta.visl.sdu.dk/visl/da/parsing/automatic/parse.php

[3] The CG3 compiler is developed by GrammarSoft ApS and supported by the University of Southern Denmark. It is open source and can be downloaded at http://beta.visl.sdu.dk/cg3.html

affecting both tokenization (name fusing and other multiple-word expressions), primary tags and secondary tags. Primary tags are tags intended to be disambiguated and evaluated, and differences in e.g. which kind of nouns are regarded as proper nouns, may therefore affect evaluation. But even secondary tags may have an adverse effect on performance. Secondary tags are lexicon-provided tags, e.g. valency and semantic tags not themselves intended for disambiguation, but used by the grammar to contextually assign primary tags. Most importantly, the gold corpus derived from the treebank does not contain semantic tags, while current DanGram rules rely on them for disambiguation. However, this is not relevant to the experiments we will be discussing in this paper - any accuracy figures are not intended to grade the performance of DanGram as such, but only to demonstrate possible performance improvements triggered by our grammar tuning. For this purpose, a certain amount of errors in the base system is desirable rather than problematic. In fact, for one of the experiments we intentionally degraded the base grammar by removing every second rule from it.

3.1 Training process and evaluation set-up

The available revised CG corpus was split randomly into 10 equal sections, reserving in turn each section as test data, and using the remaining 9 jointly as training data, a method known as 10-fold cross-validation.

For training, grammar changes (first of all, rule movements) were applied based on a performance rating of a run with the unchanged grammar (0-iteration) on the training data [4]. After a test run, the resulting, changed grammar-1 was then itself applied to the training data, and a further round of changes introduced based on the updated performance. At first, we repeated these steps until results from the test runs stabilized in a narrow F-score band. Though with certain parameter combinations this might take dozens of rounds, and though secondary, relative performance peaks were observed, we never actually found absolute maximum values beyond the 3rd iteration for either recall or precision. Therefore, most later runs were limited to 3 iterations in order to save processing time.

[4] This unchanged run also served as the baseline for our experiments (cp. dR, dP and dF in the tables).

3.2 Exploiting section structure

Constraint Grammar allows for section-grouping of rules, where the rules in each section will be iterated, gradually removing ambiguity from the input, until none of the rules in the section can find any further fully satisfied context matches. After that, the next batch of rules is run, and the first set repeated, and so on. For 6 sections, this means running them as 1, 1-2, 1-3, 1-4, 1-5, 1-6. CG grammarians use sectionizing to prioritize safe rules, and defer heuristic rules, so one obvious machine learning technique is to move rules to neighbouring sections according to how well they perform, our basic set-up being a so-called PDK-run **(P**romoting, **D**emoting, **K**illing**):**

- if a rule does not make errors or if its error percentage is lower than a pre-set threshold, promote the rule 1 section up[5]
- if a rule makes more wrong changes than correct changes, kill it altogether
- in all other cases, demote the rule 1 section down

The table below lists results (**R**ecall, **P**recision and **F**-score) for this basic method for all subsections of the corpus, with a rule error threshold of 0.25 (i.e. at most 1 error for every 4 times the rule was used). Apart from considerable cross-data variation in terms of recall improvement (dR), precision improvement (dP) and F-score improvement (dF), it can be seen that recall profits more from this setup than precision, with the best run for the former adding 0.8 percentage points and the worst run for the latter losing 0.09 percentage points.

	R	dR	P	dP	F	dF
part 1	98.11	0.22	94.6	-0.07	96.30	0.07
part 2	97.90	0.42	94.21	0.04	96.78	0.23
part 3	98.26	0.51	94.56	-0.06	96.37	0.25
part 4	97.80	0.36	93.08	-0.09	95.38	0.13
part 5	97.78	0.59	92.94	0.09	95.30	0.33
part 6	97.72	0.48	93.74	0.16	95.69	0.31
part 7	97.89	0.40	94.78	0.04	96.31	0.21

[5] First section rules can also be promoted, the effect being that they go to the head of the first section, bypassing the other rules in the section.

part 8	97.07	0.67	94.30	0.19	96.15	0.42
part 9	97.99	0.63	94.63	0.20	96.28	0.41
part 10	97.69	0.80	93.52	0.28	95.56	0.53
average	97.92	0.51	94.03	0.08	95.94	0.29

Table 1: Break-down of 10-fold cross-validation for a simple PDK run

Changing the error threshold up or down (table 2, 10-part average), decreased performance[6]:

average	R	dR	P	dP	F	dF
th=0.10	97.88	0.471	94.00	0.045	95.90	0.250
th=**0.25**	97.92	**0.509**	94.03	0.082	95.93	**0.288**
th=0.40	97.88	0.475	94.00	0.047	95.90	0.253

Table 2: Effect of changed rule error threshold (th) for a simple PDK run

We expected that iterative runs would correct initial detrimental role movements, while leaving beneficial ones in place, but for almost all parameter settings, further iterations did more harm than good. We tried to dampen this effect by reducing the rule error threshold with each iteration (dividing it by the number of iterations), but the measure did not reverse the general falling tendency of the iterated performance curve. In fact, the curve had a steeper decline, possibly because the falling threshold prevented the grammar from reversing bad rule movements.

run	0	1	2	3	4	5
th=0.25	96.12	**96.36**	96.21	96.18	96.13	**96.20**
th=*1/it	96.12	**96.36**	96.06	95.33	95.47	95.55

Table 3: F-scores for test chunk 3, per iteration

Suspecting, that hand-annotation errors in the gold corpus might cause iteration decreases by overtraining, we changed all rule-error counts by -1, among other effects permitting promoting of single-error rules, but this was overall detrimental[7].

In order to isolate the relative contributions of promoting, demoting and rule killing, these were also run in isolation:

	R	dR	P	dP	F	dF
promote	97.41	0.005	94.18	0.232	95.77	0.123
demote	97.41	0.015	94.21	**0.259**	95.77	0.127
kill	97.85	**0.440**	93.97	0.021	95.87	**0.227**

Table 4: Individual contribution of P, D and K

The results show that killing bad rules is by far the most effective of the three steps[8]. Interestingly, the three methods have different effects on recall and precision. Thus, killing bad rules prioritizes recall, simply by preventing the rules from removing correct readings. The effect of promoting and demoting almost exclusively affected precision, with demoting having a somewhat bigger effect. It should also be noted that though killing bad rules is quite effective, this does not hold for the "less bad than good" demoting category (see definition in 3.1), since killing demotable rules, too (PKK, i.e. promote-kill-kill, table 5), while marginally increasing recall, had an adverse effect on overall performance, as compared with a full PDK run. On the other hand, killing cannot be replaced by demoting, either: In a test run where bad>good rules were not killed, but instead simply demoted (PDD1) or - preferably - moved to the last section (PDD6), the expected slight increase in precision gain was more than offset by a larger decrease in recall gain. Finally, the third factor, promoting, can be shown to be essential, too, since removing it altogether (DK) is detrimental to performance.

	R	dR	P	dP	F	dF
PDK	97.92	0.509	94.03	0.082	95.93	**0.288**
PKK	**98.02**	0.611	93.86	-0.193	95.84	0.193
PDD1	97.52	0.115	94.31	**0.355**	95.89	0.239
PDD6	97.52	0.107	94.32	**0.373**	95.89	0.245
DK	97.91	0.504	94.00	0.051	95.92	0.269

Table 5: Killing instead of demoting (PKK), and demoting (PDD) instead of killing

[6] Further continuous 0.05 step variation was performed, but followed the general tendency and were left out in table 2.

[7] There was only one of the 10 sets, where error count reducing had a slight positive effect.

[8] Killed rules might be an area where human intervention might be of interest, in part because rules that do more bad than good, probably do not belong even in an untuned grammar, and in part, because a human would be able to improve the rule by adding NOT contexts etc, rather than killing it altogether.

3.3 Sorting rules

Another way of re-ordering rules is sorting all rules rather than moving individual rules. As a sorting parameter we calculated the worth W of a given rules as

$$W(rule) = G(rule)^{a} / (G(rule) + B(rule))$$

where G (=good) is the number of instances where the rule removed a wrong reading, and B (=bad) the number of instances where the rule removed a correct reading [9]. The exponent a defaults to 1, but can be set higher if one wants to put extra weight on the rule being used at all.

The most radical solution would be to sort all rules in one go, then introduce section boundaries in (six) equal intervals to prevent heuristic rules from being used in too early a pass (exploiting the 1, 1-2, 1-3 ... rule batching property of CG compilers). However, this sorting & resectioning algorithm produced poor results when used on its own - only when the original human sectionizing information was factored in by dividing rule worth by section number, was some improvement achieved (0.1 percentage points). A third option investigated was ordering rules one section at a time, which didn't help much, but was assumed to be easier to combine with rule movements in one and the same run.

	R	dR	P	dP	F	dF
resec-tioning	97.41	0.005	93.95	0.007	95.65	0.007
resect.+ /section weighti.	97.51	**0.103**	94.05	**0.106**	95.74	**0.104**
sort by section	97.44	0.033	93.98	0.031	95.67	0.031

Table 6: sorting-only performance

Putting extra weight on rule use, i.e. increasing the a exponent variable, did not increase performance, cp. the results below (with sorting performed section-wise after rule movement):

average	R	dR	P	dP	F	dF
10/10						
a=**1**	97.72	**0.312**	94.00	**0.058**	95.81	**0.173**
a=1.2	97.58	0.171	93.96	0.019	95.73	0.094

Table 7: Effect of used-rule weighting

3.4 Rule relaxation and rule strictening

The third optimization tool, after rule movement and sorting, was rule relaxation, the rationale being that some (human) rules might be over-cautious not only in the sense that they are placed in too heuristic a rule section, but also in having too cautious context conditions. A typical CG rule uses contexts like the following:

1. (-1C ART)
2. (-1 ART)
3. (*1C VFIN BARRIER CLB)
4. (*1 VFIN BARRIER CLB)
5. (*1 VFIN CBARRIER CLB)

Rule 1 looks for an article immediately to the right, while rule 3 looks for a finite verb (VFIN) anywhere to the right (*1) but with clause boundaries (CLB) as a search-blocking barrier. In both rules the 'C' means cautious, and the compiler will instantiate the context in question only if it is unambiguous. Hence, a verb like 'to house' or 'to run' that can also be a noun, can act as context once another rule has removed the noun reading. Without the C (examples 2 and 4), rules with these contexts do not have to wait for such disambiguation, and will thus apply earlier, the expected overall effect being first of all improved precision, and possibly recall, especially if the change indirectly facilitates other rules, too. BARRIER conditions work in the opposite way, they are *less* cautious, if only fully disambiguated words can instantiate them [10].

To explore the effect of rule relaxation, well-performing rules with C-contexts were duplicated [11] at the end of the grammar after stripping them of any such C-markers.

[9] What is counted here, are actual instances. Counting rule actions in isolation, i.e. what the rule would have done had it been the first to be applied, was also evaluated, but had a negative effect on almost all test subsets for both P, R and F.

[10] The same holds, in principle, for NOT contexts, but since these are mostly introduced as exceptions, their very nature is to make a rule more cautious, and most CGs will not contain examples where NOT and C are combined.

[11] The original rules were still promoted - in their original forms, on top of relaxation. Blocking the originals of relaxed-duplicated rules from promoting decreased performance.

for rules with:	R	dR	P	dP	F	dF
PDK	97.92	0.509	94.03	0.082	95.93	0.288
PDK r<1	97.86	**0.456**	94.13	0.180	95.95	0.311
PDK r<5	97.85	0.441	94.18	0.230	95.97	0.330
PDKR	97.85	0.442	94.25	**0.302**	95.65	**0.370**

Table 8: C-relaxation (added rules) instead of (pDKr), or on top of promotion (PDKr)

As can be seen, performance was clearly higher than for role movement alone, (PDKr). Setting the "well-performing"-threshold at either < 1 or < 5 errors for the rule in question, made almost no difference for recall, but showed a slight precision bias in favour of the latter. On the whole, the success of C-relaxation resides in its precision gain, which more than outweighed a moderate loss in recall.

We also experimented with relaxing such rules in situ, rather than duplicating them at the end of the grammar, but without positve effects. Similarly, no positive effect was measured when relaxing BARRIER contexts into CBARRIERs, or with combinations of C- and BARRIER-relaxation. Finally, adding in-section sorting to the C-relaxation was tried, but did not have a systematic positive effect either.

Of course, the opposite of rule relaxation, something we here will call "rule strictening" might also be able to contribute to performance, improving recall by making bad rules more cautious, waiting for unambiguous context. In this vein, we tried to add C conditions to all rules slated for demoting[12]. However, for most runs there was no overall F-score improvent over the corresponding non-stricting runs, independently of whether C-strictening was performed in situ or in combination with demoting. The only exception was PDKR(s), where stricting worked as a counter-balance to the threshold-less relaxation. As expected, recall and precision were very unequally affected by this method, and as a recall-increasing method, C-strictening *did* improve performance.

	R	dR	P	dP	F	dF
PDKR	97.85	0.442	94.25	**0.302**	95.65	0.370
PDKRs	97.88	**0.475**	94.25	0.297	96.03	**0.383**

[12] Stricting instead of killing was also tried, but without success.

PDK	97.92	0.509	94.03	**0.082**	95.93	**0.288**
PDKs	97.98	**0,571**	93.95	-0.053	95.89	0.246
PDKs in situ	97.95	0.538	93.86	-0.086	95.86	0.213
PDKr5	97.85	0.441	94.18	**0.230**	95.97	**0.330**
PDKr5s	97.89	**0.486**	94.12	0.168	95.97	0.321

Table 9: PDK rule-moving with C-relaxation (r) and strictening (s)

Combining the best stricting option with ordinary PDK and C-relaxation produced a better F-score than either method on its own, and presented a reasonable compromise on recall and precision .

3.5 PDK & rule-sorting combinations

We tested a number of further combinations of rule movement, sorting and rule relaxation/stricting, finding that sorting cannot be successfully combined with either simple rule movement (PDK, table 10) or relaxation/stricting-enhanced rule movements (PDKrs, table 11), performance being lower than for rule movement alone. If sorting is used, it should be used with the existing sectioning (sort-s) rather than resectioning (sort-S).

for rules with:	R	dR	P	dP	F	dF
PDK	97.92	**0.509**	94.03	**0.082**	95.93	**0.288**
sortPDK	97.73	**0.323**	93.96	0.014	95.80	0.162
PDKsort	97.72	0.312	94.00	**0.058**	95.81	**0.173**
sort-S + PDK	97.56	0.154	93.94	0.000	95.71	0.074
PDK + sort-S	97.41	0.006	93.96	0.012	95.65	0.009

Table 10: Effect of combining PDK and sorting, without and sort-resectioning (sort-S)

Sorting before PDK movements preserves recall better and adapts itself better to new sectioning, but the overall result is best for sorting after PDK (boldface in table 10). The only measure that could be improved by sorting, was precision in the case of sorting after a PDKr combination (bold in table 11). This effect is strongest (0.209) when resectioning is part of the sorting process (sort-S).

	R	dR	P	dP	F	dF
PDKrs	97.89	**0.486**	94.12	0.168	95.97	**0.321**
PDKrs + sort	97.85	0.444	94.07	0.117	95.92	0.274
sort + PDKrs	97.79	0.382	94.03	0.087	95.87	0.227
PDKrs + sort-S	97.47	0.064	94.15	**0.209**	95.78	0.137
sort-S + PDKrs	97.67	0.260	94.10	0.155	95.85	0.205

Table 11: Effect of combining PDKr/s and sorting

One interesting combinatorial factor is sectionizing, i.e. the creation of different or additional sections breaks in the grammar. We have already seen that sort-sectionizing (sort-S) cannot compete with the original human sectionizing, at least not with the rule sorting algorithm used in this experiment. However, sort-s is sensitive to sectionizing, too, if it is performed in connection with rule movements. To test this scenario, we introduced new start- and end-sections for rules moved to the top or bottom of the grammar, affecting especially error-free rules (top) and C-relaxed rules (bottom). The added sectioning did improve performance, but only marginally, and with no added positive effect from sorting. A more marked effect was seen when combining total C-relaxation with top/bottom-sectioning. With stricting this combination achieved the largest F-score gain of all runs (0.407 percentage points), without stricting the largest precision gain (0.318).

	R	dR	P	dP	F	dF
PDK	97.92	0.509	94.03	0.082	95.93	0.288
PDKr5s	97.89	0.486	94.12	0.168	95.97	0.321
PDKr5	97.85	0.441	94.18	0.230	95.97	0.330
PDKrSta	97.90	0.489	94.16	0.202	95.99	0.340
PDKrsS	97.93	**0.518**	94.11	0.162	95.98	0.337
PDKrS	97.89	0.480	94.18	0.227	96.00	0.349
PDKRS	97.89	0.486	94.27	**0.318**	96.05	0.399
PDKRsS	97.92	**0.518**	94.25	0.304	96.05	**0.407**
PDKRsS +sort	97.88	0.475	94.21	0.262	96.01	0.364

Table 12: PDKrs and PDKRs with new separate sections for moved start & end rules (PDKrsS)

3.6 Robustness

It is possible to overtrain a machine learning model by allowing it to adapt too much to its training data. When tuning a grammar to an annotated text corpus the risk is that rare, but possible human annotation errors will help to kill or demote a rule with very few use instances, or prevent a more frequent rule from being promoted as error-free. We were able to document this effect by comparing "corpus-true" runs with runs where all rule-error counts had been decreased by 1. The latter made the grammar tuning more robust, and led to performance improvements independently of other parameter settings, and was factored in for all results discussed in the previous sections.

Another problem is that when a large grammar is run on a relatively small one-domain training corpus, less than half[13] the rules will actually be used in any given run - which does not mean, of course that the rule will not be needed in the test corpus run. We therefore added a minimum value of 0.1 to the "good use" counter of such rules to prevent them from being weighted down as unused[14]. A corresponding minimum counter could have been added to the rule's error count, too, but given that on average rules trigger much more correct actions than errors, and assuming that the human grammarian made the rule for a reason, a small good-rule bias seems acceptable.

Finally, we had to make a decision on whether to score a rule's performance only on the instances where the rule was actually used, or whether to count instances, too, where the rule *would have* been used, if other rules had not already completely disambiguated the word in question. It is an important robustness feature of CG compilers that - with default settings - they do not allow a rule to remove the last reading of a given word, making parses robust in the face of unorthodox language use or outright grammatical errors. This robustness effect seemed to carry over into our tuned grammars - so when we tried to include 'would-discard-last-reading' counts into the rule weighting, performance decreased. The likely explanation

[13] For the 10 training corpus combinations used hear, the initial percentage of used rules was 46-47%, and considerably lower for the changed grammars in later iterations.

[14] Depending on the weighting algorithm, non-zero values are necessary anyway, on order to prevent "division-by-zero" program breakdowns.

is that rules are designed with a certain section placement in mind, so demoting rules from their current section because they would have made errors at the top of the grammar, does not make sense[15].

3.7 Grammar Efficiency

In a CG setup, grammar efficiency depends on three main parameters: First, and most obviously, it depends on the size of the grammar, and - even more - on the size of the rules actually used on a given corpus[16]. Secondly, the order of rules is also important. Thus, running efficient rules first, will increase speed, i.e. SELECT rules before REMOTE rules, short rules before long rules, high-gain/high-frequency rules before rare rules. Thirdly, a large number of sections can lead to a geometric growth in rule repetitions, and lead to a conssiderable slow down, since even if a repeated rule remains unused, it needs to run at least some negative target or context checks before it knows that it doesn't apply. In this light it is of interest, if grammar tuning has a side effect on any of these efficiency parameters. Since we have shown that neither re-sectioning nor used-rule weighting has a positive effect on perfomance, and since the relative proportion of SELECT[17] rules (SEL% in table 13) remained fairly constant, tuning is neutral with regard to the second and third parameters.

	rules	used	killed	promote (use)	demote (use)	SEL %
0	4840	2278	-	-	-	38.5
1	4734	2157	105	4581-45%	153-51%	38.2
2	4724	2163	9	3676-49%	90-73%	37.8
3	4701	2051	22	2273-46%	97-57%	37.3
4	4687	2135	13	2984-50%	100-60%	37.5
5	4678	1987	8	2008-49%	87-61%	36.3

Table 13: PDK rule use statistics,
for 10-3 training corpus (Fmax=96.36 at iteration 1)

There was, however, a falling tendency in the number of used rules with increasing iterations, in part due to rule-pruning by killing, but probably also to the promotion of safe rules that could then "take work" from later rules. For the first 2 iterations, where optimal performance usually occured, this amounts to 6-7% fewer rules.

The better-performing PDKRsS method led to a much smaller reduction in active rules (2-3%, table 14), because of the added relaxed rules that contributed to improved precision by cleaning up ambiguity after ordinary rules. Also, for the same reason, the absolute number of rules increased considerably, and because even unused rules have to be checked at least for their target condition, there actually was a 9% increase in CPU usage.

	rules	used	killed	promote (use %)	demote (use %)	SEL %
0	4840	2278	-	-	-	38.5
1	7625	2232	105	4581-45%	153-35%	38.0
2	7715	2204	21	3676-46%	84-43%	38.0
3	7821	2209	21	7481-29%	44-43%	38.0
4	7831	2217	9	7608-29%	52-44%	37.7
5	7837	2194	12	7722-28%	47-30%	38.0

Table 14: PDKRsS rule use statistics,
for 10-3 training corpus (Fmax=96.43, iteration 3)

3.8 Smaller-scale grammars

In this paper, we have so far discussed the effect of tuning on full-size, mature Constraint Grammars, determining which parameters are most likely to have a positive effect. In quantitative terms, however, the improvement potential of a smaller-scale, immature grammar is much bigger. We therefore created an artificially reduced grammar by removing every second rule from the original grammar, on which we ran the PDK+relaxation/stricting setup that had performed best on the full grammar, with optional pre- and postsorting.

	R	dR	P	dP	F	dF
original	97.41	-	93.95	-	95.65	-

[15] More specifically, it would make sense only in one scenario - section-less sorting of all rules, which proved to be an unsuccessful strategy for other reasons.

[16] Of course, independently of rule number, the disambiguation load of a corpus remains the same, and hence the number of times some rule removes a reading. However, fewer rules used means that superfluous rules could be removed from grammar, rather than trying to match their targets and contexts in vain.

[17] A SELECT rule is more efficient, because it can resolve a 3-way ambiguity in one go, while it will take 2 REMOVE rules to achieve the same.

grammar						
untuned 1/2 gr.	97.48	.	85.55	-	91.12	-
PDKr1s	97.59	0.113	86.23	0.474	91.44	0.318
PDKr1sS	97.48	0.222	85.88	0.327	91.41	0.282
PDKr5s	97.56	0.083	86.26	0.708	91.56	0.436
PDKr5sS	97.73	**0.247**	86.19	0.638	91.59	0.469
PDKRs	97.52	0.045	87.84	2.289	92.43	1.303
DKr1s	97.57	0.095	86.00	0.449	91.12	0.295
DKr5s	97.52	0.040	86.45	0.906	91.65	0.529
DKR	97.54	0.066	87.90	2.345	92.47	1.343
DKRs	97.52	0.037	87.96	**2.417**	92.42	**1.369**
DKRsS	97.92	**0.441**	85.36	-0.185	91.21	0.086
DKRs + sort	97.54	0.062	87.87	2.330	92.45	1.329

Table 15: Effects on half-sized grammar

Like for the original grammar, PDK performed best without sorting. However, a number of performance differences can be noted. First, performance maxima were achieved later, often on the third iteration rather than the first, as was common for the original grammar. Second, as might be expected, F-scores improved 4 x more in absolute, and 2 x more in relative terms, than for the full grammar. More surprisingly, the gain is entirely due to precision gains, with a small fall in recall for most runs[18]. This can probably be explained by the fact that a Constraint Grammar is in its essence reductionist - it reduces ambiguity. Inactivating part of the rules, will simply leave more ambiguity (i.e. lower precision), but not necessarily have a corresponding influence on recall, since recall depends more on the quality of the individual rule. Given this dominating importance of precision, we tried to create a precision bias by inactivating the recall-favoring choices of stricting (PDKr) and rule-killing (PDr), but for the incomplete grammar reducing recall did not automatically mean increased precision, and these combinations did not work. Surprisingly, and contrary to what was expected from the full-grammar runs, the most beneficial measure was to inactivate promoting (DKrs), and to create maximally many relaxed rules (DKRs), by removing the relaxation threshold, allowing all rules with C-conditions to relax as long as their original versions did more good than bad. Adding new top/bottom-sections produced the highest recall gains (0.441 for DKRsS), but these did not translate into corresponding F-score gains.

The iteration profile for the succesful DKR run does not show the falling oscillation curve for F-scores seen for PDK runs (table 16). Rather, there is a shallow-top maximum stretching over serveral iterations, and than a slow fall-off with late oscillation. In terms of efficiency, the iteration pattern is also quite flat, with a fairly constant SELECT-rule percentage, and a slowly falling number of used rules, with relaxed-duplicated rules compensating for the disappearance of killed rules and demoted rules.

	rules	used	killed	demote (use)	SEL%	F-score
0	2420	1383	-	-	37.7	91.55
1	3011	1670	66	100-93%	35.9	92.70
2	3821	1661	40	120-77%	36.2	92.73
3	3012	1639	23	94-83%	36.3	92.74
4	3936	1630	8	83-82%	36.6	92.75
5	3936	1624	5	73-74%	36.5	92.71

Table 16: DKR rule use statistics,
for 10-3 training corpus on reduced grammar

4 Conclusion

In this paper, we have proposed and investigated various machine learning options to increase the performance of linguist-written Constraint Grammars, using 10-fold cross-validation on a gold-standard corpus to evaluate which methods and parameters had a positive effect. We showed that by error-rate-triggered rule-reordering alone (promoting, demoting and killing rules), an F-score improvement of 0.29 could be achieved. With an F-score around 96% this corresponds roughly to a 7.5 % lower error rate in relative terms. However, we found that a careful balance had to be struck for individual rule movements, with a demoting threshold of 0.25% errors being the most effective, and that general performance-driven rule sorting was less effective than threshold-based individual movements. Likewise, the original human grammar sectioning and rule order is important and could not be improved by adding new sectioning, or even by in-section rule sorting.

[18] The only recall-preserving combination was DKr, i.e. without promoting and without stricting.

Apart from rule movements, rule changes were explored as a means of grammar optimization, by either increasing (for well-performing rules) or decreasing (for badly performing rules) the amount of permitted ambiguity in rule contexts. Thus, removing C (unambiguity) conditions was beneficial for precision, while adding C-conditions ("stricting") improved recall. Finally, section-delimiting of moved top- and bottom rules also helped. Altogether, the best combination of these methods achieved an average F-score improvement of 0.41 percentage points (10 percent fewer errors in relative terms). For a randomly reduced, half-size grammar, F-score gains are about three times as high - 1.36 percentage points or 15% in relative terms, an important difference being that for the mature grammar recall improvement contributed more than recall, while gains in the reduced grammar were overwhelmingly based on precision.

Obviously, the grammar tuning achieved with the methods presented here does not represent an upper ceiling for performance increases. First, with more processing power, rule movements could be evaluated against the training corpus individually and in all possible permutations, rather than in-batch, eliminating the risk of negative rule-interaction from other simultaneously moved rules[19]. Second, multi-iteration runs showed an oscillating performance curve finally settling into a narrow band *below* the first maximum (usually achieved already in iteration 1 or 2, and never after 3). This raises the question of local/relative maxima, and should be further examined by making changes in smaller steps. Finally, while large scale rule reordering is difficult to perform for a human, the opposite is true of rule killing and rule changes such as adding or removing C-conditions. Rather than kill a rule outright or change *all* C-conditions in a given rule, a linguist would change or add individual context conditions to make the rule perform better, observing the effect on relevant sentences rather than indirectly through global test corpus performance measures. Future research should therefore explore possible trade-off gains resulting from the interaction between machine-learned and human-revised grammar changes.

[19] With over 4,000 rules and a 3-iteration training run taking 30 minutes for most parameter combinations, this was not possible in our current set-up.

References

Eckhard Bick, Heliana Mello, A. Panunzi and Tommaso Raso. 2012. The Annotation of the C-ORAL-Brasil through the Implementation of the Palavras Parser. In: Calzolari, Nicoletta et al. (eds.), Proceedings LREC2012 (Istanbul, May 23-25). pp. 3382-3386. ISBN 978-2-9517408-7-7

Eckhard Bick. 2011. A Barebones Constraint Grammar, In: Helena Hong Gao & Minghui Dong (eds), Proceedings of the 25th Pacific Asia Conference on Language, Information and Computation (Singapore, 16-18 December, 2011). pp. 226-235, ISBN 978-4-905166-02-3

Eckhard Bick & Marcelo Módolo. 2005. Letters and Editorials: A grammatically annotated corpus of 19th century Brazilian Portuguese. In: Claus Pusch & Johannes Kabatek & Wolfgang Raible (eds.) Romance Corpus Linguistics II: Corpora and Historical Linguistics (Proceedings of the 2nd Freiburg Workshop on Romance Corpus Linguistics, Sept. 2003). pp. 271-280. Tübingen: Gunther Narr Verlag.

Eckhard Bick. 2003. Arboretum, a Hybrid Treebank for Danish, in: Joakim Nivre & Erhard Hinrich (eds.), Proceedings of TLT 2003 (2nd Workshop on Treebanks and Linguistic Theory, Växjö, November 14-15, 2003), pp.9-20. Växjö University Press

Fred Karlsson, Atro Voutilainen, Juha Heikkilä and Arto Anttila. 1995. Constraint Grammar: A Language-Independent System for Parsing Unrestricted Text. Natural Language Processing, No 4. Mouton de Gruyter, Berlin and New York

Torbjörn Lager. 1999. The µ-TBL System: Logic Programming Tools for Transformation-Based Learning. In: Proceedings of CoNLL'99, Bergen.

Nikolaj Lindberg, Martin Eineborg. 1998. Learning Constraint Grammar-style Disambiguation Rules using Inductive Logic Programming. COLING-ACL 1998: 775-779

Lluís Padró. 1996.. POS Tagging Using Relaxation Labelling. In: Proceedings of the 16th International Conference on Computational Linguistics, COLING (Copenhagen, Denmark). pp. 877--882.

Eirikur Rögnvaldsson. 2002. The Icelandic µ-TBL Experiment: µ-TBL Rules for Icelandic Compared to English Rules. Retrieved 2013-05-12 from [http://hi.academia.edu/EirikurRognvaldsson/Papers]