Enhancing ENIGMA Given Clause Guidance

Jan Jakubův and Josef Urban[⋆]

Czech Technical University in Prague, Prague, Czech Republic

Abstract. ENIGMA is an efficient implementation of learning-based guidance for given clause selection in saturation-based automated theorem provers. In this work, we describe several additions to this method. This includes better clause features, adding conjecture features as the proof state characterization, better data pre-processing, and repeated model learning. The enhanced ENIGMA is evaluated on the MPTP2078 dataset, showing significant improvements.

1 ENIGMA: Efficient Learning of Given Clause Guidance

State-of-the-art saturation-based automated theorem provers (ATPs) for first-order logic (FOL), such as E [5], are today's most advanced tools for general reasoning across a variety of mathematical and scientific domains. Many ATPs employ the *given clause algorithm*, translating the input FOL problem $T \cup \{\neg C\}$ into a refutationally equivalent set of clauses. The search for a contradiction is performed maintaining sets of *processed* (P) and *unprocessed* (U) clauses. The algorithm repeatedly selects a *given clause* g from U, extends U with all clauses inferred with g and P, and moves g to P. This process continues until a contradiction is found, U becomes empty, or a resource limit is reached. The search space of this loop grows quickly and it is a well-known fact that the selection of the right given clause is crucial for success.

ENIGMA [4] stands for *E*fficient lear*N*ing-based *I*nference *G*uiding *MA*chine that steers clause selection in saturation-based ATPs like E. ENIGMA is based on a simple but fast *logistic regression* algorithm effectively implemented by the LIBLINEAR open source library [2]. In order to employ logistic regression, first-order clauses need to be translated to fixed-length numeric *feature vectors*. ENIGMA uses (top-down-)oriented term-tree walks of length 3 as *features*. For example, a unit clause "$P(f(a,b))$" contains only features "(P,f,a)" and "(P,f,b)" (see [4, Sec. 3.2] for details). Features are enumerated and a clause C is translated to the feature vector φ_C whose i-th member counts the number of occurrences of the i-th feature in clause C. We also count top-level literal symbols (positive or negative) and we unify variables and Skolem symbols.

In order to train an ENIGMA *predictor* $\mathcal{E}$, all the given clauses $\mathcal{C}$ from a set of previous successful proof searches are collected. The given clauses used in

[⋆] This work was supported by the ERC Consolidator grant no. 649043 *AI4REASON*, and by the Czech project AI&Reasoning CZ.02.1.01/0.0/0.0/15_003/0000466 and the European Regional Development Fund.

the proofs are classified as positive ($\mathcal{C}^+ \subseteq \mathcal{C}$) and the remaining given clauses as negative ($\mathcal{C}^- \subseteq \mathcal{C}$). The clause sets ($\mathcal{C}^+, \mathcal{C}^-$) are turned into feature vector sets (Φ^+, Φ^-) using a fixed *feature enumeration* π. Then a LIBLINEAR *classifier* w (a *weight vector*) is trained on the classification (Φ^+, Φ^-), classifying each clause as *useful* or *un-useful* for the proof search. The classifier w and enumeration π give us a predictor $\mathcal{E} = (w, \pi)$ which is used to guide next proof searches in combination with other E heuristics. Thanks to the fast feature extraction mechanism and the fast (linear) evaluation of the features in a particular learned model, there is no slowdown of the given clause loop. In fact, ENIGMA is faster than some of the more advanced hand-programmed E evaluation heuristics [3]. The training speed allows fast MaLARea-style [8] iterative loop between ATP proving and re-learning [4, Sec. 5.1].

2 Enhanced ENIGMA Features and Classifiers

This section briefly describes improvements from the previous ENIGMA version [4].

Sparse Features Encoding. Previously, ENIGMA was tested only on the CASC 2016 AIM benchmark [7] which contains only about 10 different symbols. Using a straightforward dense encoding yielded feature vectors of size 10^3, because ENIGMA features are triples of symbols. Such exhaustive enumeration of feature vectors was too big for larger symbol signatures. The term-tree walks features are, however, relatively sparse: symbols are typically applied only to a small number of other symbols. Hence we switched to a sparse encoding, using only the features which actually appear in the training data. This significantly reduced the feature vector sizes while preserving the predicting accuracy. This alone allows us to test ENIGMA on Interactive Theorem Proving (ITP) benchmarks which tend to have much larger signatures.

Conjecture Features. Second, for AIM we initially did not consider conjectures to be a part of the model. Hence the same clauses were being recommended in every possible AIM proof search. This can work (similarly to the *hint* method [9]) on benchmarks of very related problems (such as AIM), but hardly on large ITP-based formal libraries, which are much more heterogeneous. To overcome this, we embed the *conjecture features* in the feature vectors. For a clause C, instead of using the vector φ_C of length n (where n is the number of different features appearing in the training data), we use a vector (φ_C, φ_G) of length $2n$ where φ_G contains the features of the conjecture G. For a training clause C, G corresponds to the conjecture of the proof search where C was selected as a given clause. When classifying a clause C during a proof search, G corresponds to the conjecture currently being proved.

Horizontal Features. The previously described term-tree walk features of length 3, can be seen as *vertical* chains in the term-tree. In order to provide a more accurate representation of clauses by feature vectors, we additionally introduce *horizontal* features. For every term $f(t_1, \ldots, t_n)$, we introduce the feature $f(s_1, \ldots, s_n)$ where s_i is the top-level symbol of the subterm t_i. Our feature enumeration π is extended with the horizontal features from the training data, and the feature vectors again record the number of occurrences of each such feature in a clause. For example, the unit clause "$P(f(g(a), g(a)))$" is characterized by one occurrence of horizontal features "$P(f)$" and "$f(g, g)$", and by two occurrences of "$g(a)$".

Static Features. We have also introduced *static* clause features, based on existing E's *feature vectors* used for subsumption indexing [6]. The first three features are signature independent and consist of (1) the clause length, and (2-3) the counts of positive/negative literals. For each signature symbol f, we also use (4-5) the number of occurrences of f in positive/negative literals, (6-7) the maximal depth of any occurrence of f in positive/negative literals.

E uses *clause evaluation functions*, which assign a numeric *weight* to every clause, to select the next given clause. The weight is computed for every generated clause and the clause with the smallest weight is selected. The ENIGMA weight function for predictor $\mathcal{E}$ assigns a smaller weight to clauses positively classified by $\mathcal{E}$. Previously, we had to combine this weight with the clause length to prevent E from generating very long positively classified clauses. With the new static features this is no longer necessary.

Accuracy-Balancing Boosting. The training data produced by previous proof searches are typically unbalanced in the number of positive and negative samples. Usually there are many more negatives and predictors trained directly on such data usually mis-classify positive examples much more than the negative ones. Previously, we used trivial boosting methods, like repeating positive samples ten times, to correct this behavior. Now we use the following iterative boosting method. Given training data $(\mathcal{C}_0^+, \mathcal{C}_0^-)$ we create an ENIGMA predictor $\mathcal{E}_0$, test $\mathcal{E}_0$ on the training data, and collect the positives mis-classified by $\mathcal{E}_0$. We then repeat (*boost*) the mis-classified positives in the training data, yielding updated $(\mathcal{C}_1^+, \mathcal{C}_0^-)$ and an updated predictor $\mathcal{E}_1$. We iterate this process, and with every iteration, the accuracy on the positive samples is increased, while the accuracy on the negatives is typically decreased. We finish the boosting when the positive accuracy exceeds the negative one.

Directed Looping. An E *strategy* S is a collection of E's options that influence the proof search. An important part of every strategy S is the collection of clause evaluation functions used for given clause selection. Given an ENIGMA predictor $\mathcal{E}$ trained on proofs done with S, we can either (1) use $\mathcal{E}$ alone to select the given clause, or we can (2) combine $\mathcal{E}$ with other evaluation functions from S. In (2) we by default select half of the given clauses by $\mathcal{E}$, and the other half by the original selection functions from S. This gives rise to two new E strategies, denoted (1) $S \odot \mathcal{E}$, and (2) $S \oplus \mathcal{E}$. In practice, there is usually no clear winner between the two as there are problems solved by the first one but not the other, and the other way round.

Given a set of training problems P and a strategy S, we can run S for each problem from P, extract training samples $\mathcal{C}_0 = (\mathcal{C}_0^+, \mathcal{C}_0^-)$, and create the predictor $\mathcal{E}_0$. Then we can evaluate both ENIGMA strategies $S \odot \mathcal{E}_0$ and $S \oplus \mathcal{E}_0$ on P and obtain additional training samples. This yields new training samples $\mathcal{C}_1 \supseteq \mathcal{C}_0$ and a new predictor $\mathcal{E}_1$. The predictor $\mathcal{E}_1$ usually combines the strengths of both

article	total	autos	ENIG	ENIG+	article	total	autos	ENIG	ENIG+
compts	23	7	7	+0.0%	filter	65	6	7	+16.7%
enumset1	96	86	86	+0.0%	orders	61	28	36	+28.6%
pre	37	21	22	+4.8%	wellord1	59	27	35	+29.6%
relset	32	20	22	+10.0%	yellow19	38	12	18	+50.0%
funct	235	160	185	+15.6%	waybel	174	42	74	+76.2%

Table 1. Best-performing ENIGMA models for selected articles.

ENIGMA strategies $S \odot \mathcal{E}_0$ and $S \oplus \mathcal{E}_0$. This looping process can be iterated and after several iterations, strategies $S \odot \mathcal{E}_i$ and $S \oplus \mathcal{E}_i$ start performing almost equally, typically outperforming the initial strategy S on P. This way we produce an enhanced single strategy. Currently we just iterate this process three times. Future work includes adaptive termination criteria based, for instance, on a respective performance of $S \odot \mathcal{E}_i$ and $S \oplus \mathcal{E}_i$.

3 Experimental Evaluation

The previous version of ENIGMA was evaluated with a single E strategy on the CASC 2016 AIM benchmark [7]. Here we additionally evaluate on the Mizar MPTP2078 [1] dataset (MZR) as an example ITP benchmark. This section presents three experiments designed to (1) evaluate the effect of conjecture features on prediction accuracies, (2) evaluate other enhancements on the training data, and to (3) test the ability of ENIGMA to generalize to similar problems.

We use both the AIM and MZR benchmarks to evaluate the effect of conjecture features on predictor accuracy. We first evaluate 10 E strategies on the benchmarks and construct 10 different ENIGMA predictors. We measure the *10-fold cross-validation* accuracy, where the data are divided into 10 subsets (*folds*) with 9 folds used for training and one fold left aside for evaluation. As expected, adding the conjecture features helps on the MZR data, improving an averaged 10-fold cross-validation accuracy from 88.8% to 91.5%. On AIM this improves from 76.3% to 77.8%. We explain this by the AIM problems having more similar conjectures. The feature vector sizes vary from 60 to 130 on AIM and from 2300 to 22000 on MZR.

Next we evaluate the effect of the other enhancements on MZR. The MZR problems are naturally clustered into 26 categories, based on the original Mizar article which was the source for the Mizar-to-FOL translation. Hence the problems within each category are typically related and use similar symbols. The E auto mode contains more than 60 strategies and we select the ten best-performing on MZR problems. This gives us a portfolio of 10 well-performing strategies, denoted *autos*, and we attempt to further improve this portfolio using the ENIGMA predictors.

We train a separate predictor $\mathcal{E}_{S,A}$ for every strategy S from *autos*, and for every article (category) A, using all the enhancements from Section 2. This gives us 10 ENIGMA strategies per article (currently, for predictor $\mathcal{E}_{S,A}$ we simply take $S \oplus \mathcal{E}'_{S,A}$ after 3 iterations of looping). Now, on each article A, we can compare

portfolio	solved	E%	autos%	autos+	autos-
E (auto-schedule)	1343	+0%	-3.8%	+25	-79
autos (10)	1397	+4.0%	+0%	+0	-0
ENIGMA (62)	1450	+7.9%	+3.7%	+103	-50

Table 2. Overall portfolio improvement.

the portfolio of the ten *autos* with the portfolio of the ten ENIGMA strategies trained on A. This tells us how ENIGMA strategies perform on the training data. Table 1 summarizes the results for several articles. All the E runs were performed with 10 seconds time limit per problem (precisely, each portfolio strategy runs for 1 second on a problem). The *total* column shows the number of problems in the article A, *autos* is the number of problems solved by *autos*, the *ENIG* column presents the number of problems solved by the ENIGMA strategies, and finally *ENIG+* summarizes the gain in percentage of the ENIGMA strategies on *autos*. On the training data, ENIGMA strategies were able to solve all the problems solved by *autos* and also additional problems. The gain in percentage varies from 0% to a surprisingly high value of 76.2%, with 15.7% on average.

The next experiment is designed to test the ability of ENIGMA predictors to generalize, and to enrich an already well-performing portfolio. From the previous experiment, we have altogether $26 * 10$ ENIGMA strategies. The strategies trained on article A have been already evaluated on A, and thus we can compute their greedy cover on A. To reduce the number of strategies, we take only the strategies in the greedy cover of some article A. This reduction gives us a portfolio of 62 ENIGMA strategies, which can now be compared with the performance of the ten *autos* on the whole MZR benchmark. We use the time limit of 300 seconds per problem, equally divided among the strategies in a portfolio.

The results are presented in Table 2. We additionally evaluate E in the *auto-schedule* mode, which is a state-of-the-art general-purpose native strategy scheduler of E. The numbers in parentheses indicate the number of problems in a given portfolio. The *E%* column measures the gain on *auto-schedule* in percents, *autos%* measures the gain on *autos*, and *autos+* and *autos-* state the difference between the problems solved by the respective portfolio and by *autos*. The *autos* portfolio outperforms *auto-schedule* even though *auto-schedule* contains all the strategies from *autos* (and more). This is because we have constructed *autos* from the ten strategies which perform (greedily) best on the MZR benchmark. We can see that the ENIGMA strategies solve 3.7% more problems than the initial *autos* strategies, and 7.9% more than *auto-schedule*. This means that the ENIGMA strategies are capable of generalizing to problems similar to the training problems with a measurable improvement. Without the enhancements from Section 2, we had been able to improve single strategies, but we were much less successful in enriching a well-performing portfolio of diverse strategies. The new ENIGMA enhancements are instrumental for achieving this.

4 Software Distribution

We added support for ENIGMA to E Prover[1] and we developed a simple Python module `atpy` for construction of ENIGMA models and evaluation of E strategies. The module is available at our web page[2] and it contains an example with instructions (see `examples/README.ENIGMA.md`) on how to use `atpy` for ENIGMA experiments. The required binaries for LIBLINEAR and the extended E Prover are provided (for x86) together with example benchmark problems and E strategies.

The `atpy` module is capable of (1) evaluation of E strategies on custom benchmark problems, (2) extraction of training samples from E outputs, (3) construction of ENIGMA models with boosting and looping, and of (4) construction of E strategies enhanced with ENIGMA predictors. The ENIGMA models are stored in the directory `Enigma` in separate subdirectories. The model directory contains files (1) `model.lin` with LIBLINEAR weight vector w and (2) `enigma.map` with feature enumeration π. Horizontal features are represented by strings of the form "$\mathtt{f.s_1.\cdots.s_n}$", and vertical features by "$\mathtt{s_1:s_2:s_3}$". The extended E supports our clause evaluation function `Enigma` which takes a relative model directory as an argument. Our E extension additionally includes a feature extraction tool `enigma-features` required by `atpy`.

References

1. Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., Urban, J.: Premise selection for mathematics by corpus analysis and kernel methods. J. Autom. Reasoning **52**(2), 191–213 (2014). https://doi.org/10.1007/s10817-013-9286-5
2. Fan, R., Chang, K., Hsieh, C., Wang, X., Lin, C.: LIBLINEAR: A library for large linear classification. Journal of Machine Learning Research **9**, 1871–1874 (2008). https://doi.org/10.1145/1390681.1442794, http://doi.acm.org/10.1145/1390681.1442794
3. Jakubův, J., Urban, J.: Extending E prover with similarity based clause selection strategies. In: CICM. Lecture Notes in Computer Science, vol. 9791, pp. 151–156. Springer (2016)
4. Jakubův, J., Urban, J.: ENIGMA: Efficient learning-based inference guiding machine. In: CICM. Lecture Notes in Computer Science, vol. 10383, pp. 292–302. Springer (2017)
5. Schulz, S.: E - A Brainiac Theorem Prover. AI Commun. **15**(2-3), 111–126 (2002)
6. Schulz, S.: Simple and efficient clause subsumption with feature vector indexing. In: Automated Reasoning and Mathematics. Lecture Notes in Computer Science, vol. 7788, pp. 45–67. Springer (2013)
7. Sutcliffe, G.: The 8th IJCAR automated theorem proving system competition - CASC-J8. AI Commun. **29**(5), 607–619 (2016). https://doi.org/10.3233/AIC-160709, http://dx.doi.org/10.3233/AIC-160709
8. Urban, J., Sutcliffe, G., Pudlák, P., Vyskočil, J.: MaLARea SG1 - Machine Learner for Automated Reasoning with Semantic Guidance. In: IJCAR. pp. 441–456 (2008)

[1] https://github.com/ai4reason/eprover/tree/ENIGMA
[2] https://github.com/ai4reason/atpy

9. Veroff, R.: Using hints to increase the effectiveness of an automated reasoning program: Case studies. Journal of Automated Reasoning **16**(3), 223–239 (1996). https://doi.org/10.1007/BF00252178