Applying Machine Learning to Enhance Optimization Techniques for OWL Reasoning

Razieh Mehri* and Volker Haarslev

Concordia University, Montréal, Québec, Canada
r_mehrid@encs.concordia.ca
haarslev@cse.concordia.ca

Abstract. Various (tableau) optimization techniques have been integrated into OWL reasoners to speed up reasoning. Many of the techniques rely on heuristics that have been manually fine tuned for achieving a good performance but might fail dramatically when encountering ontologies exhibiting unexpected design patterns. A typical example are heuristics applied to disjunctions in order to select a disjunct to be added to the tableau. Evidences indicate that the order of selecting disjuncts can have a significant impact on reasoning speed. Our approach presented in this paper applies machine learning to make the selection process more effective and removes the need for manual fine tuning. We extended the OWL reasoner JFact accordingly to control the disjunct selection process. We demonstrate that one can successfully learn to choose a disjunct based on the most effective heuristic method. As a first step we focused on propositional SAT testing. Our results show that machine learning can speed up JFact by one to two orders of magnitude.

Keywords: OWL reasoning · Description logic reasoning · Machine learning · Tableau optimization techniques

1 Introduction and Related Work

A large amount of research has been devoted to the design of tableau optimization techniques for Web Ontology Language (OWL) reasoning. Heuristic guided optimization techniques try to decrease the size of the search space by making better decisions while dealing with non-deterministic expansions (disjunctions). While these methods try to improve the optimization techniques, learning from these techniques has not yet been considered for helping OWL reasoners to make the best decision when applying a non-deterministic rule.

Among the many optimization techniques, semantic branching is directly affected by the non-determinism caused by disjunctions; nevertheless, semantic branching was developed to reduce the non-deterministic cost of syntactic branching as a part of traditional tableau algorithms.

In our enhanced version of semantic branching we apply machine learning to choose a disjunct that could have a drastic impact on reasoning speed. One

* Corresponding author

heuristic technique suggested for this purpose is to use the *MOMS* heuristic that looks for a disjunct with the maximum number of occurrences in disjunctions of minimum size [4]. This technique could be more effective with backjumping which is an optimized backtracking variant [2]. Moreover, a combination of characteristics of axioms such as size, maximum quantifier depth, frequency of concepts occurring in axioms can help to make better decisions [16].

Semantic branching uses the Davis-Putnam-Logemann Loveland (DPLL) algorithm that splits branches by making them disjoint from each other. DPLL is used to solve the satisfiability problem (SAT) in propositional logic. Therefore, learning to apply the most proper order for variables while solving a SAT problem helps with the order of applying variables in semantic branching.

A SAT solver's runtime varies based on the given instance and the heuristic algorithm used for choosing variables in each decision level. In order to solve a SAT problem efficiently, the approach in [7] applies reinforcement learning to select the branching heuristic with a maximum long-term reward to give the most proper order for variables with the least overall cost. In [12], authors use a different learning technique called multinomial logistic regression to solve the Quantified Boolean Formulas (QBF) that is a more expressive generalization of SAT. The solution was only suggested for QBF expressions with binary clauses.

In our work, we employ multinomial logistic regression too. Our approach, however, applies machine learning to semantic branching in OWL reasoners and not to SAT solvers. Although, propositional logic is less expressive than Description Logic (DL) [1], yet we believe that a significant percentage of OWL ontologies contain concept descriptions resembling propositional logic. Moreover, we use a variety of available branching heuristics, from traditional to new ones such as [12], which uses *VSIDS* along with other heuristics derived from *VSIDS*. Also, our method does not restrict the number of clauses or variables in logic expressions.

There also exist methods that do not consider heuristics while detecting the best possible expansion. An optimization technique called learning-based disjunct was introduced in [14]. This method reduces the expansions of inherently clash generating disjuncts. It orders the disjuncts based on the characteristics they share with already expanded clash free disjuncts. Further, some reasoners dramatically benefit from caching [11]. Caching can be used indirectly for the (un)satisfiability status of a (sub/super) concepts in disjunctions [3, 6]. Caching is used with quantifiers and in this case, the status of generated successors can be cached for look-alike concepts.

In order to integrate our approach into JFact, for each clause in the reasoner's input, the order of variables is changed. The new order is specified based on the order given by solving an input as a SAT problem using machine learning; in other words, JFact uses our fully trained model to find the new order. Thus, the clauses in each branching level of reasoning have a new order that could lead to an earlier solution. The speed improvement is caused by the reduced occurrence of backjumping due to the changes in (semantic) branching.

The remainder of the paper is as follows: Section 2 provides a brief background on semantic branching and its relation to DPLL as well as backjumping, whereas Section 3 describes how machine learning is used in our algorithm together with branching heuristics. Section 4 describes our experiments.

2 Reasoning Optimizations

Testing the satisfiability of a concept in tableau-based systems might be costly due to non-deterministic expansions. OWL/DL reasoners try to avoid such a cost by using optimization techniques that can save memory and/or time. Our learning technique has an (in)direct impact on two of these optimization techniques: semantic branching and backjumping.

Semantic Branching: Traditional tableau algorithms use a not very efficient algorithm called syntactic branching. Due to the redundant search space, which syntactic branching might explore, new reasoners apply semantic branching instead [4].

Backjumping: This is a search reduction technique which uses dependency lists to avoid redundant backtracking in completion graphs. Despite backtracking, backjumping does not just backtrack from a clash to the most recent expanded disjunction, but it also considers whether a disjunction is related to a clash by creating a dependency set for each disjunct and labeling the concepts with sets containing the non-deterministic choices in each branch [2].

3 Learning to Select Variables

3.1 Branching Heuristics

As already noted, the order of applying variables in semantic branching can significantly affect the resulting runtime. Based on different branching heuristics, variables can have different scores. These branching heuristics try to prune the search tree by targeting specific variables and clauses.

The equation below determines the score of each variable ($Score(Var)$) by considering the score of its literals in each branch, so it could balance between the branches to increase the variable's score. The first branch considers the positive from $score(Lit^+)$ and the second branch considers the negation from $score(Lit^-)$: $Score(Var) = score(Lit^+) + score(Lit^-)$

Choosing the variable with the highest score ($Score(Var)$) may save time by leading us to an earlier solution. Moreover, by knowing the scores of both literals or branches ($score(Lit^+)$ and $score(Lit^-)$), one can also choose between the branches. In our implementation, we are currently only concerned about changing the order of selecting variables in the reasoning process.

Below are the well-known branching heuristics used in our implementation.

First literal The first literal occurring in the boolean formula is returned.

MOMS The literal with the maximum number of occurrences in the clauses with minimum size is considered.

$$MOMS(l) = \text{occurrences of } l \text{ in minimum size clauses.}$$

MOMSF It is the alternative of *MOMS*. If $f(x)$ is the number of occurrences of the variable x in the clauses of minimum size, we return the variable maximizing $(f(Lit^+) + f(Lit^-)) * 2^k + (f(Lit^+) * f(Lit^-))$ where k is a defined constant. [5] indicated some factors which lead to choosing the best priority function.

MAXO The literal with the maximum number of occurrences in the boolean formula has the greatest score.

$$MAXO(l) = \text{number of occurrences of } l \text{ in the formula.}$$

JW The Jeroslaw-Wang chooses the literal that maximizes the following equation where n_j is the number of literals in the clause C_j:

$$JW(l) = \sum_{j, l \in C_j} 2^{-n_j}$$

JW2 The two sided Jeroslaw-Wang is the same as JW, but to choose a variable it considers its score by adding the score of its positive and negative literals together.

DLCS It counts the number of clauses in which the literal and its negation occur in and assigns them respectively with CP and CN:

$$DLCS(Var) = CP(Var) + CN(Var)$$

The variable with maximum $DLSC$ is selected. If $CP \geq CN$, the positive form of variable (l) is chosen and if $CP < CN$, the negative form $(\neg l)$ is chosen. $DLSC$ is faster than JW.

POSIT it is similar to $DLCS$, but the search is only done on clauses with minimum size.

DLIS It is the same as $DLCS$, but it chooses the maximum value among all CP and CN values; then, it picks the variable with the maximum value. $DLIS$ performs faster than $DLCS$ and on some benchmarks it performs twice as fast as JW.

3.2 Learning Model

A learning method is applied to assign a heuristic method to each logic expression based on its features. In this study, we use logistic regression.

In comparison with linear regression, which predicts a continuous dependent variable for each independent variable (instance), logistic regression predicts a categorical dependent variable (class) for each independent variable.

The model built for logistic regression considers the dependent variable as a conditional probability given as $P(Y = 1|X = x)$ that indicates the probability of choosing a specific class (identified as 1) for an instance x. If the resulting probability is more than 0.5, then the instance belongs to class 1; otherwise, it belongs to class 0 [13].

The represented model for linear regression is a linear equation that relates independent variables ($\mathbf{x}$ with n features $x_1, x_2, ..., x_n$) to dependent variables ($h(\mathbf{x})$):

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = \sum_{i=0}^{n} w_i x_i$$

Note that $x_0 = 1$ and $\mathbf{w}$ stands for parameters or weights.
Using the same model for logistic regression leads to:

$$P(Y = 1|\mathbf{x}) = \mathbf{w}^T\mathbf{x}$$

which is not promising, since the probability should be bounded between 0 and 1. Therefore, the modified equation represented for logistic regression is:

$$P(Y = 1|\mathbf{x}) = \frac{e^{\mathbf{w}^T\mathbf{x}}}{1 + e^{\mathbf{w}^T\mathbf{x}}}$$

Multinomial Logistic Regression If the number of classes to predict is more than two (several branching heuristics), then the logistic regression technique is called multinominal logistic regression and it is defined by:

$$P(Y = H_j|\mathbf{x}) = \frac{e^{\mathbf{w}_{H_j}^T\mathbf{x}}}{\sum_{H'} e^{\mathbf{w}_{H'}^T\mathbf{x}}}$$

where $\sum_H P(H|\mathbf{x}) = 1$, $H = H_1, H_2, ..., H_j, .., H_k$. Notice that for all k classes, the probability of an instance labeled as each of those classes will be obtained; finally, among the obtained probabilities the maximum one defines the class that the instance belongs to.

To obtain the best value for parameters ($\mathbf{w}$), the *Newton-Raphson* method is used to maximize the approximation of the predictor function. To obtain the best ($\mathbf{w}$), *Newton-Raphson* uses *Hessian* and *Gradient* that are the second and first derivative of the error function ($J(\mathbf{w})$), respectively. In our experiment, the following formula is repeated 11 times to obtain the best $\mathbf{w}$ ($\mathbf{w}^0$ is initiated with a vector of 0s):

$$\mathbf{w}^{i+1} = \mathbf{w}^i - \frac{J'(\mathbf{w}^i)}{J''(\mathbf{w}^i)}$$

A set of benchmarks is used as training data for this method. The training data from the benchmarks contains instances ($\mathbf{x}$). To obtain a class or heuristic (y_i) of each instance (x_i), all heuristics are applied to the instances and for each instance the heuristic that runs fastest will be considered as the class of that instance. Afterwards, the training data pairs (x_i, y_i) are ready to be used for the *Newton-Raphson* method. While computing $\mathbf{w}_{H_j}$, the training instances with class H_j will be identified as class 1 and others are 0.

The error function ($J(\mathbf{w})$) in logistic regression is defined as:

$$J(\mathbf{w}) = -\sum_{i=1}^{m} y_i log(h(x_i)) + (1 - y_i) log(1 - h(x_i))$$

To avoid overfitting, the regularized term $\lambda \sum_{i=1}^{m} w_i^2$ is added to the cost function (*Hessian* and *Gradient* formulas will also be changed accordingly) where λ is a regularization parameter.

Table 1. Training data chosen from SATLIB (number of samples)

Library	JNH	AIM	PARITY	Flat	CBS	RTI	BMS	UF3SAT	Others	Overall
Training samples	47	24	9	79	623	24	80	367	147	1400
Satisfiable	14	16	9	79	623	24	80	279	107	1231
Unsatisfiable	33	8	0	0	0	0	0	88	40	169

Table 2. Overall training data attributes

Number of variables			Number of clauses		
Min	Max	Mean	Min	Max	Mean
1	558	103	1	1801	434

4 Empirical Evaluation

For our implementation and experiments, we use JFact[1], which is a Java port of the FaCT++[2] reasoner [15]. JFact is an open source Java-based tool which makes it easily modifiable and portable to all platforms.

4.1 Training Data

The training data used in our experiment are files in DIMACS CNF format. The DIMACS format was introduced for encoding the input of SAT solvers, in which each literal is a number and each clause is a line containing literals followed by zero. Our training data are from different benchmarks available on SATLIB.[3] Those benchmarks are: JNH, AIM, PARITY, Flat, CBS, RTI, BMS, and Uniform Random. The available benchmarks in the website are limited to the specific range of variable and clause numbers; for example, there are only few CNF files with less than 50 variables and all of the files have 20 variables. To avoid this restriction, we added about 147 additional CNF files. These additional files contain small CNF inputs with between 1 to 50 variables. The final training data contains 1400 files including both satisfiable and unsatisfiable CNFs. Tables 1 and 2 show some characteristics of the training data from SATLIB.

In this experiment, we only deal with propositional logic input. Therefore, the training data contains DIMACS CNF format files to be solved with a SAT solver. We used a standard SAT solver based on the DPLL algorithm.

To verify whether providing more training data can give us more accurate results, we use a learning curve to show how increasing the number of training data can increase/decrease the accuracy of our classifier. As shown in Fig. 1, we are confident that adding more training data will not change the accuracy of the classifier. The ten features, which have been used for our experiments, are

– Number of variables (*var*)

[1] `https://sourceforge.net/projects/jfact/`
[2] `https://code.google.com/archive/p/factplusplus/`
[3] `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

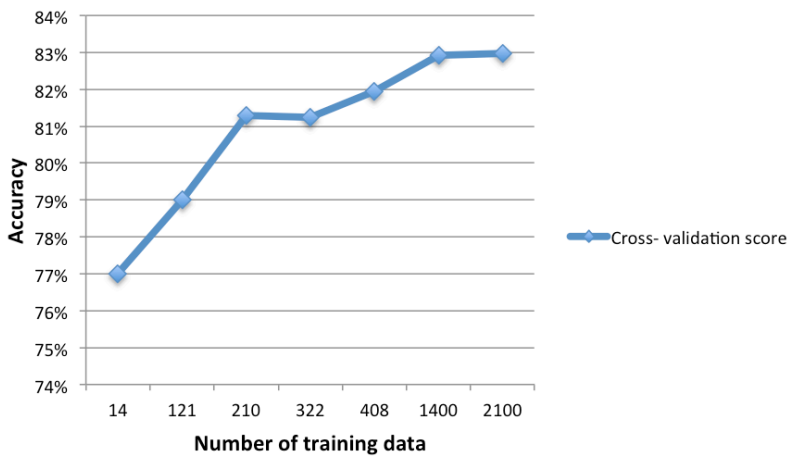

Fig. 1. Cross-validation learning curve

Table 3. Number and percentage of training samples and the improved runtime for each heuristic

Heuristics	First Literal	DLSC	DLIS	MOMS	MOMSF	POSIT	ZM	JW	JW2
Count	92	159	129	90	292	170	111	137	220
Percentage	6.57%	10.71%	9.21%	6.42%	20.85%	12.14%	7.92%	9.78%	15.71%

- Number of clauses (cls)
- $\frac{var}{cls}$
- $(\frac{var}{cls})^2$
- $(\frac{var}{cls})^3$
- Fraction of binary clauses
- Fraction of ternary clauses
- Fraction of horn clauses (clauses with exactly one positive literal)
- Number of positive literals
- Number of negative literals

These features are chosen based on the characteristics that play a main role when using heuristics for solving SAT problems. Selecting too many features may lead to overfitting depending on the number of training samples. Regularization helps to prevent overfitting if there are too many —and possibly irrelevant —features [10]. Therefore, in this study, we use regularization to allow as many features as possible before overfitting.

The nine branching heuristics used for our experiments are: *FirstLiteral*, *MOMS*, *MOMF*, *JW*, *JW2*, *POSIT*, *ZM*, *DLSC*, and *DLIS*. Table 3 shows for each heuristic on how many training samples (and for what percentage) it resulted in the shortest runtime compared to the other heuristics.

Although more heuristic methods have been developed recently, none of them is the most effective solution for all benchmarks. For example, *MOMS* may only

be good for logic expressions with many binary or unary clauses [8], but when considering all possible input and including features such as the number of unary and binary clauses, *MOMS* may lead to very effective results. Therefore, in our study the branching heuristics were chosen in such a way to consider CNF files with different shapes. Most of our sample tests, which are introduced later, choose *MOMSF* based on their features and the application of the learning algorithm, which determines the probability of the heuristics.

Based on the algorithm, for each training data, the selected branching heuristic is used until the solver process is finished. The same applies for the sample tests.

The sample cases chosen as input are OWL files in conjunctive normal form. Depending on the complexity of the input (it does not matter if it is already in conjunctive normal form), JFact builds a directed acyclic graph (DAG) structure of that input [9]. The built DAG could be explicit or implicit based on the structure of the input. Changing the order in the DAG structure can be done while building the DAG.

Our samples are OWL files created from CNF DIMACS format files in which a concept, which is targeted for checking its satisfiability, is equal to the CNF. Since the target concept contains the whole CNF, JFact builds an explicit structure from the OWL file; therefore, targeting specific points of the DAG structure can be done easily since all levels of the DAG structure are directly involved in building the concept. More consideration is needed when the DAG is not explicit since one needs to figure out which levels of the DAG are directly involved in building a concept, i.e., the process of checking satisfiability of that concept. As an example, let us consider the following CNF DIMACS format data.

```
p cnf 4 3
2 3 -4 1 0
-4 0
2 3 4 0
```

Our OWL generator builds an OWL file from the CNF DIMACS file and adds a fresh concept named D that is equivalent to the given CNF. Therefore, we get the following description logic axiom.

$$\begin{aligned} D \equiv\ & (A \sqcup B \sqcup C \sqcup \neg E) \sqcap \\ & \neg E \sqcap \\ & (B \sqcup C \sqcup E) \end{aligned}$$

The DAG structure built by JFact for the above axiom is

```
1 *TOP*
2 primconcept(urn:jfact#temp) [= 1
3 concept(ontology:#D) = 10
4 primconcept(ontology:#A) [= 1
5 primconcept(ontology:#B) [= 1
```

Table 4. Samples (runtimes in milliseconds)

No.	Runtime (without learning)	Runtime (with learning)	No. of backjumps (without learning)	No. of backjumps (with learning)	Runtime speedup	Backjump reduction	No. of variables	No. of clauses	Sat/ Unsat status
1	22,651	10,569	1,892,308	834,000	2.14	2.26	60	250	Sat
2	25,247	3,699	2,252,016	267,111	6.82	8.43	65	257	Sat
3	597,447	534,735	56,537,985	50,330,597	1.11	1.12	60	160	Unsat
4	52,989	31,587	4,380,675	2,609,718	1.67	1.67	75	325	Unsat
5	20,229	10,222	1,771,462	789,925	1.97	2.24	59	261	Unsat
6	16,770	5,511	1,019,601	350,294	3.04	2.91	90	354	Unsat
7	7,268	1,800	709,942	158,874	4.03	4.46	45	160	Sat
8	106,941	4,393	7,699,075	271,423	24.34	28.36	75	305	Sat
9	18,779	800	1,212,269	32,067	23.47	37.80	75	325	Sat
10	19,493	9,511	1,593,995	740,132	2.04	2.15	59	243	Unsat
11	6,866	1,641	677,743	136,166	4.18	4.97	45	160	Sat
12	5,744	753	551,661	49,356	7.62	11.17	34	155	Sat
13	110,952	15,244	6,627,625	850,760	7.28	7.8	75	343	Unsat

```
6 primconcept(ontology:#C) [= 1
7 primconcept(ontology:#E) [= 1
8 and 7 -4 -5 -6
9 and -5 -6 -7
10 and -7 -8 -9
```

Here, D is a newly added defined *concept*, which stands for a non-primitive concept, and it is linked to node 10. Other concepts called A, B, C, E are defined as *primconcept* (stands for primitive concept). They are used to build D and are linked to the concept $\top$ (Top) as node 1.

4.2 Results

Our experiment contains 40 sample tests with the number of clauses between 150 and 400 and the number of variables between 35 and 90. Half of the samples are satisfiable and half are unsatisfiable. The experiments were performed on a MacBook Pro with 2.2 GHz Intel Core i7, RAM 16 GB 1600 MHz DDR3 using a correspondingly modified version of JFact. We carried out a 7-fold cross validation on 1400 training sets with a learning accuracy of 83%.

The performance of 13 out of 40 of our sample tests are shown in Table 4. These samples with their characteristics give an adequate representation (more

similar in their characteristics compared to other samples) of all 40 samples. For each sample the table lists the original runtime (no learning), the improved runtime (with learning), the number of original backjumps (no learning), the number of reduced backjumps (with learning), the runtime speedup (defined as original runtime divided by improved runtime), the backjump reduction (defined as the number original backjumps divided by the number of reduced backjumps), and also the number of clauses, variables, and the SAT status. Table 4 also demonstrates that there is a direct relationship between the speed improvement and the reduction in the number of backjumps. For example, for the sample number 8, the number of backjumps went from 7,699,075 down to 271,423 (reduction of 28) while the runtime improved from 106,941 milliseconds to 4,393 milliseconds (speedup of 24).

The overall performance of the algorithm is given in Table 5. It lists the maximum, minimum and mean runtime improvement in milliseconds and the corresponding speedup factors. This shows that applying learning improves the speed of satisfiability checking by one to two orders of magnitude.

Moreover, the percentage of improvement based on (un)satisfiability suggests that the learning technique is more successful for satisfiable cases. The rate of success for unsatisfiable and satisfiable input are 53.01% and 78.25%, respectively.

Since we want to achieve a significant runtime improvement (e.g., at least 10%), we mostly consider sample tests that perform in a sufficient amount of time (e.g., at least several seconds) such that the improvement is obvious. For example, the samples with below 50 variables and 100 clauses mostly perform in less than one second; therefore, improving their speed is not our primary concern.

On the other hand, the sample number 3 from Table 4 (with 60 variables and 160 clauses) has a initial runtime of 597,447 milliseconds and the runtime is improved by 62,712 milliseconds after learning. Therefore, the percentage of improvement is 10.49%, which is still considered as good. Moreover, for the CNF files with more than 100 variables and 400 clauses, our experiments require several minutes of runtime. For example, if a sample test is executed in 5 minutes and the improvement is only 10 seconds, then the improvement percentage is only 3 percent which we do not consider as significant. That is also one of the reasons, that in our 40 successful sample tests we only included very few with long running times since the learning improvement is not always very significant.

Our implementation does not improve the runtime for every input. For one out of every five input files (20%) the runtime is mostly unchanged or increased by less than 10%. For example, in one sample test, the runtime without learning is 502,131 milliseconds. After learning, the runtime has increased by 2.43%, which we consider as a tolerable performance loss.

To reduce the number of cases without a significant speedup, we believe it is a good idea to discover specific CNF patterns where a learning improvement is not expected. For example, in one of the identified patterns for every clause there also exist clauses with the exact same variables of that clause but every

Table 5. Speed improvement

Improvement	**Max**	**Min**	**Mean**
in milliseconds	102,548	1,892	18,500
Speedup Factor	311.2	1.11	14.3

other possible combinations of their literals. An example of such pattern is the following description logic axiom.

$$D \equiv (A \sqcup E \sqcup F) \sqcap (A \sqcup \neg E \sqcup F) \sqcap (A \sqcup E \sqcup \neg F) \sqcap (A \sqcup \neg E \sqcup \neg F) \sqcap (\neg A \sqcup E \sqcup F) \sqcap (\neg A \sqcup \neg E \sqcup F) \sqcap (\neg A \sqcup E \sqcup \neg F) \sqcap (\neg A \sqcup \neg E \sqcup \neg F) \sqcap (B \sqcup C) \sqcap (B \sqcup \neg C) \sqcap (\neg B \sqcup C) \sqcap (\neg B \sqcup \neg C)$$

Note that the features such as the number of variables and clauses are close to the successful samples.
Finding these patterns can also help us to categorize OWL input to enhance the reliability of our future implementation, which will also be extended for more expressive OWL input.

5 Conclusion and Future Work

In this paper, we focused on improving one of the well known optimization techniques for OWL reasoners called semantic branching. Even though semantic branching is guaranteed to avoid redundant search space occurring in traditional syntactic branching, machine learning techniques also help to further decrease redundant search space exploration. The task is possible by learning new orderings for applying variables in each branching level. Our algorithm has demonstrated a significant improvement in our sample tests.

As part of our future work, we started to expand our approach beyond propositional description logic and integrate the proper treatment of universal, existential quantifiers, and qualified number restrictions in our machine learning approach. For instance, we started working on strategies to apply machine learning in a similar way while also considering interactions between OWL axioms.

Similar strategies could be applied to simplify OWL expressions and prepare them for machine learning. Later, for training purposes, OWL expressions could be treated as propositional logic or Quantified Boolean Formulas (QBF) so they can take advantage of available training sets to be solved by QBF or SAT solvers. Moreover, as already mentioned in the previous section, categorizing OWL input based on its shape and pattern could be also helpful for further exploration. Finally, we plan to consider other learning techniques such as Naive Bayes and k-Nearest Neighbor based on the features and their correlation.

References

1. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, 2nd edn. (2007)
2. Baker, A.B.: Intelligent backtracking on constraint satisfaction problems: experimental and theoretical results. Ph.D. thesis, University of Oregon (1995)
3. Donini, F.M., Massacci, F.: Exptime tableaux for $\mathcal{ALC}$. Artif. Intell. 124(1), 87–138 (Nov 2000)
4. Freeman, J.W.: Hard random 3-SAT problems and the Davis-Putnam procedure. Artificial intelligence 81(1-2), 183–198 (1996)
5. Freeman, J.W.: Improvements to propositional satisfiability search algorithms. Ph.D. thesis, University of Pennsylvania (1995)
6. Haarslev, V., Möller, R.: High performance reasoning with very large knowledge bases: A practical case study. In: IJCAI 2001. pp. 161–168 (2001)
7. Lagoudakis, M.G., Littman, M.L.: Learning to select branching rules in the DPLL procedure for satisfiability. Electronic Notes in Discrete Mathematics 9, 344–359 (2001)
8. Maandag, P., Barendregt, H., Silva, A.: Solving 3-SAT. Bachelor's thesis, Radboud University Nijmegen (2012)
9. Nebot, V., Berlanga, R.: Efficient retrieval of ontology fragments using an interval labeling scheme. Information Sciences 179(24), 4151–4173, doi:10.1016/j.ins.2009.08.012 (2009)
10. Ng, A.Y.: Feature selection, L 1 vs. L 2 regularization, and rotational invariance. In: Proceedings of the twenty-first international conference on Machine learning. p. 78. ACM (2004)
11. Patel-Schneider, P.F.: DLP system description. In: In Proc. Description Logics (DL)-98. pp. 78–79 (1998)
12. Samulowitz, H., Memisevic, R.: Learning to solve QBF. In: In Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI'07). vol. 7, pp. 255–260 (2007)
13. Shalizi, C.: Advanced data analysis from an elementary point of view. Cambridge University Press (2013)
14. Sirin, E., Grau, B.C., Parsia, B.: From wine to water: Optimizing description logic reasoning for nominals. In: In: Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006). pp. 90–99 (2006)
15. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. In: International Joint Conference on Automated Reasoning. vol. 4130 of LNCS, pp. 292–297. Springer (2006)
16. Tsarkov, D., Horrocks, I., Patel-Schneider, P.F.: Optimizing terminological reasoning for expressive description logics. Journal of Automated Reasoning 39(3), 277–316 (2007)