2015 IEEE Symposium Series on Computational Intelligence

Evolving Workflow Graphs Using Typed Genetic Programming

Tomáš Křen, Martin Pilát
Charles University in Prague
Faculty of Mathematics and Physics
Malostranské nám. 2/25
Prague, Czech Republic
Email: Tomas.Kren@mff.cuni.cz
Email: Martin.Pilat@mff.cuni.cz

Roman Neruda
Institute of Computer Science
Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2
Prague, Czech Republic
Email: roman@cs.cas.cz

***Abstract*—When applying machine learning techniques to more complicated datasets, it is often beneficial to use ensembles of simpler models instead of a single, more complicated, model. However, the creation of ensembles is a tedious task which requires a lot of human interaction and experimentation. In this paper, we present a technique for construction of ensembles based on typed genetic programming. The technique describes an ensemble as a directed acyclic graph, which is internally represented as a tree evolved by the genetic programming. The approach is evaluated in a series of experiments on various datasets and compared to the performance of simple models tuned by grid search, as well as to ensembles generated in a systematic manner.**

I. Introduction

There have been many machine learning methods proposed in the last decades, and new ones are appearing every day. Each of these methods naturally performs differently for different datasets and even for different instances in the same datasets. It is well-known that the combination of several simple methods into a single more complicated ones can improve the results significantly. Such combinations of methods are also known as *ensembles*. Ensemble methods have gained a lot of attention recently after they have been able to win the famous Netflix prize, and they are also successful in other types of competitions. For example, almost all competitions organized by the popular Kaggle community are won by ensemble models. Among the simplest ensemble methods would be the voting used for classification tasks. In this case, several models are trained on the same data, and when a new instance shall be classified, the models vote for the predicted class, i.e. each model predicts a class and the most common one is used as the prediction of the ensemble.

Typically, the creation of an ensemble requires a lot of non-trivial work – tuning of parameters of different models, combination of suitable models, and so on. This is a tedious task depending on the experience of the person performing it. In this work, we proposed an approach based on typed genetic programming which is able to automatically create ensembles of machine learning methods. The approach utilizes the fact that each ensemble can be described as a directed acyclic graph and uses the typed genetic programming to evolve such graphs. The types ensure that the data flowing in the graph are consistent and that the whole graph makes sense from the data-mining point of view.

In the next chapter, we describe the work related to the paper at hand, then we proceed to describe the proposed algorithm, which is then evaluated on four classification datasets from various application areas in Section IV. Some preliminary results and ideas from this paper – namely concerning the systematic generation of graphs – were already published as an extended abstract [1], however, the vast majority of information in this paper is new.

II. Related Work

Most work regarding machine learning workflows focuses on the creation of ontologies which describe the individual parts of the workflow and allow for a more formal approach to the problem. For example, Panov *et al.* [2] designed such an ontology called OntoDM. The ontology contains definitions of basic data-mining entities and also supports definition of more complex concepts. Diamantini *et al.* [3] use a different ontology to create a composition procedure that generates knowledge discovery workflows which are later ranked using various criteria.

From a more search-based point of view, Kazík *et al.* [4], experiment with the combination of various preprocessing and machine learning techniques, and they compare the results. However, the workflows in this paper are rather simple linear sequences of several preprocessing techniques followed by a single data-mining one. Recently, Folino and Pisani [5] used genetic programming to evolve function which would combine the outputs of several machine learning techniques into an ensemble.

The technique presented in this paper follows a slightly different path than the ontology based techniques. Instead of an ontology, which describes how the workflow can work, we used types in genetic programming which encode similar information, however, it also allows for natural search and optimization of the possible workflows. We have already published this basic idea in an abstract [1], however, in that work we did not use any genetic programming, and the types were used only to guide a systematic A*-based search of all possible workflows. This work is largely extended in this paper

978-1-4799-7560-0/15 $31.00 © 2015 IEEE
DOI 10.1109/SSCI.2015.200

and the types and genetic programming are described here in much more detail for the first time.

There are several works [6], [7], [8] using typed GP with parametric polymorphism. All those works, as far as we know, evaluate their systems on rather simple benchmark problems in means of time needed to evaluate one individual (e.g. *even-parity* problem), whereas the problem of ensemble construction is very time consuming, and thus it is closer to real world applications. Here, we also present a novel approach to population initialization and individual mutation based on uniform generating of polymorphic trees which is rather complicated due to expressive power of parametric polymorphic types. We also demonstrate a technique of lifting natural numbers into type system in order to tightly control a correct structure of individuals. This technique is easily applicable in other typed GP problems.

III. Our Approach

A. Workflow graph described as a polymorphously typed tree

A classifier may be seen as a black box transforming data (D) to labeled data (LD). Such black boxes may be combined into more complicated ensembles. One way of combining them is by splitting input data into several parts which are fed into different classifiers. Finally, outputs of those classifiers may be put together by some kind of merge (e.g. voting). A slightly more complicated ensemble is depicted on the figure 1.A where a preprocessing node is also present.

One can see that such ensemble naturally forms a directed acyclic graph (DAG). Another important observation is the fact that not every DAG represent a correct ensemble; the inputs must be connected to outputs in a way that respects the data types flowing through them.

In order to describe, generate, and manipulate such correct DAGs we use typed syntactic trees. Though, we are not aiming exclusively at completeness (describing all the correct DAGs), but more at sufficient compromise among description simplicity, symmetry handling and completeness.

Let us describe our approach by means of terminal and function set from which the individual trees are build. Terminal nodes of trees correspond to specific machine learning methods such as a single classifier, preprocessing, clustering, or voting method. Function nodes represent operations combing several DAGs into one DAG.

For the sake of clarity we first show how the example ensemble from the figure1.A is constructed. We can demonstrate it by successive decomposition[1] of the DAG into the syntactic tree representing the DAG (see Fig. 1).

In part (A), we see the whole DAG with data flow arrows. In (B), the first top-level decomposition is shown; here the whole DAG is decomposed into three DAGs connected serially. This operation is called ens_1 (which is described in greater detail below). In (C) the decomposition is performed by another serial operation called $split$. There is also a parallel operation provided by $cons$ function used in part (D). In (E), a situation after skipping few similar steps is depicted where the whole tree representing the DAG is obtained.

[1]Note that we show the process of decomposition for illustrative purposes; during the evaluation of tree individuals the process of building a DAG from a tree has the opposite direction.

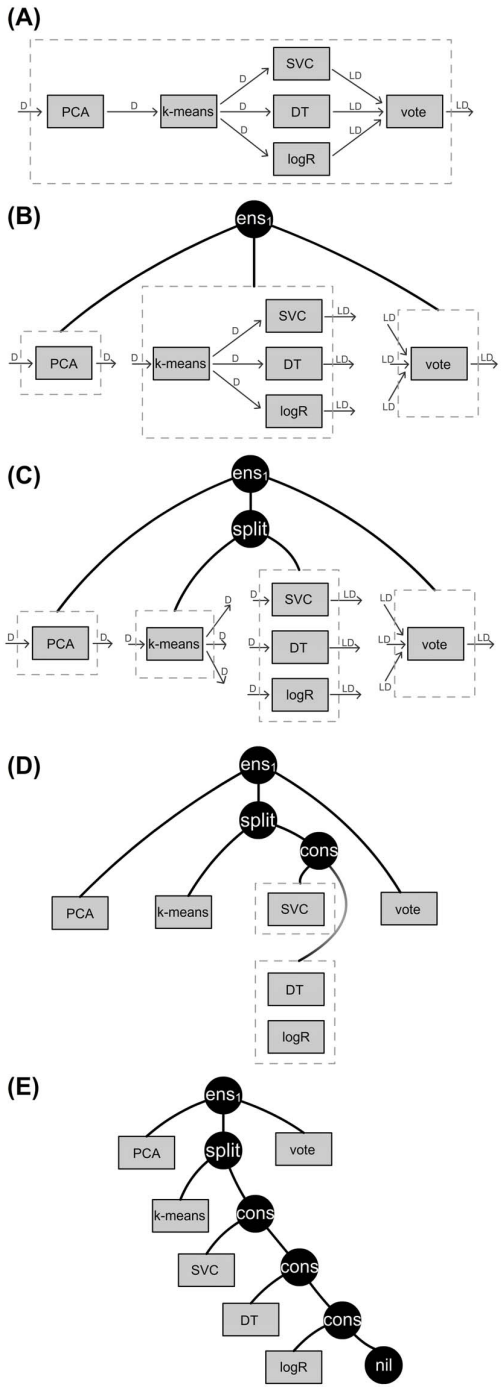

Fig. 1. Example of decomposition of a DAG into a syntactic tree

In the process of generating DAGs by composing smaller ones, there is one important issue that must be addressed: When we want to serially compose two DAGs, the number of outputs of the first DAG must be the same as the number of inputs of the second one.

We cope with this issue on the type level by using a type system with parametric polymorphism. Namely, we use a data type representing a list of a fixed length (vector). It is denoted as (**V** a n), where a is a type variable standing for the type of an element, and n is an auxiliary type variable standing for the type encoding the length of such list. The construction of numbers (on the type level) representing list lengths is performed in a traditional way: There is an atomic type **0** standing for 0, and an auxiliary parametric type (**S** n) standing for $n+1$. Thus, for example, the term (**V Int** (**S** (**S** (**S 0**)))) denotes a list of three integers.

In order to be able to construct fixed sized lists, the terminal set T contains a polymorphic constant *nil* : (**V** a **0**) which stands for an empty list, and the function set F contains a polymorphic function *cons* : $a \times$ (**V** a n) $\rightarrow$ (**V** a (**S** n)) which constructs a new list of length $n+1$ from a head element (first input) and a tail list of length n (second input).

During DAG composition, not only the numbers of outputs and inputs must match, but the same hold for their types. We handle this constrain by using parametric type (**Dag** a b) standing for a DAG with input type a and output type b (where a or b may be a fixed length list). For example (**Dag D** (**V LD** (**S** (**S** (**S 0**))))) is a type of a DAG that has one input node for data and three outputs nodes for labeled data (e.g., the middle DAG on the figure 1.B has this type).

Let us look on types of the machine learning methods in the terminal set. *Preprocessors* (e.g. PCA) have type (**Dag D D**). *Splitters* (e.g. k-means) have type (**Dag D** (**VD** (**S** (**S** n)))), since we want them to split the data to at least two outputs, but we do not want to specify the number of outputs by hand. *Classifiers* (e.g. decision tree) have type (**Dag D LD**). And *mergers* (e.g. voting) have type (**Dag** (**V LD** (**S** (**S** n))) **LD**).

The function set F contains three DAG constructing operations ens_0, ens_1 and $split$. The $split$ operation serially connects a *splitter* with n parallel DAGs packed in a fixed size list, each having the *classifier type* (**Dag D LD**) (cf. Fig. 1); i.e. $split$ has the type (**Dag D** (**V D** n)) $\times$ (**V** (**Dag D LD**) n) $\rightarrow$ (**Dag D** (**V LD** n)). The ens_0 operation serially connects a result of the $split$ operation (which may be called *splitter into **LD***) with a $merger$ resulting in a construction of an ensemble with the *classifier type*; i.e. ens_0 has the type (**Dag D** (**V LD** n)) $\times$ (**Dag** (**V LD** n) **LD**) $\rightarrow$ (**Dag D LD**). The ens_1 operation is a variant of ens_0 enriched with additional argument for *preprocessor* (cf. Fig. 1); the ens_1 operation has type (**Dag D D**) $\times$ (**Dag D** (**V LD** n)) $\times$ (**Dag** (**V LD** n) **LD**) $\rightarrow$ (**Dag D LD**).

B. Generating typed trees with parametric polymorphism

Here we present a novel approach to uniformly generating trees in the type system involving parametric polymorphism such as Hindley-Milner type system [9]. More specifically, our method uniformly picks a random tree for a given (goal) type and tree size (*see Algorithm 1*). A pair $(goalType, treeSize)$ is called *query*. In order to make the generating of one tree very fast, an auxiliary data structures are used to hold information about numbers of trees for particular queries. The important benefit of this approach is that once the auxiliary data are computed the generating procedure is always able to generate a well-typed individual without a need of backtracking from dead ends – it generates trees in one pass. Moreover, the auxiliary data information can be directly reused among many runs of evolution, and even among different problems if they share the same symbol set $T \cup F$.

The auxiliary data are organized in the following way. There is a **QueryResult** holding the information for a specific query (*see Algorithm 2*). **QueryResult** also has a list of **SubResult**s holding the same kind of information but for a more specific *subquery* specified by a particular root *symbol* for the generated trees and by a particular *size profile* – that is a specific choice of sizes for subtrees respecting the query tree size.

In order to compute the auxiliary information effectively in a *dynamic programming* fashion, we need to store more detailed information than just a single number for each query. This is mainly because a $goal$ type of a query may be a general type including a type variable (e.g. (**Dag** a b)), but we want to count to this query even trees with a more specific type (e.g. (**Dag D LD**)). To cope with this we use a key-value map[2] with types as keys and numbers as values to store information about numbers for types with greater or equal specificity (i.e. the **this**.$nums$ field in the ***QueryResult***).

Let us comment on some notions from the pseudocode. A *substitution* is denoted as σ. It is a mapping of type variables to more specific types. A substitution can be understood as a function, so we can apply substitution σ to to type t to obtain a more specific type $\sigma(t)$. A substitution can be also composed by $\circ$ operator (since it is a function). In the pseudocode, there is a widely used function **MGU**$(type_1, type_2)$ that finds the *most general unifier*, that is the most general substitution σ such that $\sigma(type_1) = \sigma(type_2)$ if it exists, or it *fail*s.

A detailed explanation of our generating algorithm is beyond the scope of this paper, however we provide simplified but rather detailed pseudocode that is hopefully self explanatory for the reader familiar with type systems and with the involved techniques such as substitutions and unifications. The presented algorithm is a generalization for polymorphously typed GP of our previous work on individual generating introduced in [10].

The generating method can be straightforwardly used for population initialization and for mutation. In the initialization phase, for each tree to be generated we first uniformly select a $treeSize \in \{1, \ldots, 15\}$ and then generate it by ***generateOne***(**Dag D LD**, $treeSize$). For mutation, a random subtree is picked and replaced by a newly generated alternative of the same size.

Our method also evolves parameters for the machine learning methods. In the initialization procedure, the trees are generated without parameters which are generated in the post-processing phase of initialization. More detailed discussion of parameter tuning (including a special parameter mutation) continues bellow.

We use simple typed crossover that uniformly chooses one of all possible pairs of subtrees (one from each parent) with

[2]In the pseudocode we use a map.**merge**$(key, newVal, op)$ method for manipulating a map which inserts $newVal$ if the map does not contain key, or updates the value to $op(oldVal, newVal)$.

Algorithm 1: Uniformly generating one tree individual.

```
function generateOne(Type goal, Int treeSize)
    qResults ← query(goal, treeSize)
    if qResult.num = 0 then
        return null
    else
        i ← select uniformly from {0, num − 1}
        for subRes ∈ qResult.subResults do
            if i < subRes.num then
                sons ← [ ]
                σ ← subRes.σ
                for (sonGoal, sonSize) ∈ subRes.qs
                do
                    goal_2 ← σ(sonGoal)
                    son ←
                    generateOne(goal_2, sonSize)
                    σ ← MGU(son.type, goal_2) ∘ σ
                    sons.add(son)
                sym ← subRes.sym
                return new Tree(sym, sons, σ(goal))
            i ← i − subRes.num
```

Algorithm 2: Construction of a new query result.

```
function query(Type goal, Int treeSize)
    qResult ← qResults.get((goal, treeSize))
    if qResult = null then
        qResult ← new QueryResult(goal, treeSize)
        qResults.put((goal, treeSize), qResult)
    return qResult
constructor QueryResult(goal, treeSize)
    this.subResults ← [ ]
    this.nums ← {}
    for sym ∈ T ∪ F do
        (argTypes, outType) ← freshenTVars(sym)
        σ ← MGU(goal, outType)
        if σ ≠ fail then
            n ← sym.arity
            sizeProfiles ← allProfiles(treeSize, n)
            for sp ∈ sizeProfiles do
                qs ← [ ]
                for i ∈ {0, . . . , n − 1} do
                    qs.add((argTypes[i], sp[i]))
                sr ← new SubResult(goal, sym, qs, σ)
                if sr.num ≠ 0 then
                    this.subResults.add(g)
                    for (t, num) ∈ sr.nums do
                        nums.merge(t, num, +)
    this.num = sum(nums)
```

Algorithm 3: Construction of a new query subresult.

```
constructor SubResult(goal, sym, qs, σ)
    this.qs ← qs
    this.σ ← σ
    this.nums ← nums(goal, sym, qs, σ, 1)
    this.num ← sum(nums)
function nums(goal, sym, qs, σ, acc)
    if qs.isEmpty() then
        return { σ(goal) : acc }
    else
        result ← {}
        (sonGoal, sonSize) ← qs.getHead()
        goal_2 ← σ(sonGoal)
        sonResult ← query(goal_2, sonSize)
        for (t, num) ∈ sonReult.nums do
            σ_2 ← MGU(t, goal_2) ∘ σ
            acc_2 ← num × acc
            rest ← qs.getTail()
            nums_2 ← nums(goal, sym, rest, σ_2, acc_2)
            for (t_2, num_2) ∈ nums_2 do
                nums_2.merge(t_2, num_2, +)
        return result
```

the same type (so that the offspring is well-typed after the crossover) and the chosen pair of subtrees is swapped.

IV. Experiments

To assess the performance of the above described technique, we run a series of experiments on four different datasets obtained from the UCI machine learning repository [11]. In this section, we first describe the datasets, then we discuss the settings of the evolutionary algorithm and the evaluation metrics, and finally we provide the results of the experiments followed by discussion.

A. Datasets

We used four different datasets: `winequality` [12], `wilt` [13], `ml-prove` [14], and `magic` [15]. All of them are classification datasets with real attributes. The `winequality` dataset contains 4,898 instances with 11 attributes describing physical and chemical properties of white wine (e.g. pH, density, acidity, ...). The goal is to predict the quality of the wine on a scale 1-9. The goal of the `wilt` dataset is to classify image segments into two classes, – either contains a deceased tree, or contains anything other – based on 6 real-valued attributes obtained by image analysis. There are 4,839 in the dataset (we merged the training and testing set from the UCI repository). The two classes are strongly imbalanced (261 vs 4,578 instances). The `ml-prove` dataset consists of 6,118 instances (we have again merged the training, testing, and validation sets from the UCI repository), each with 51 attributes describing various feature of automated theorem proving tasks. The goal is to predict, which of five heuristics would perform the best (i.e. prove the theorem in the shortest time), there is also an option that none of the heuristics is able to solve the given problem, thus the dataset asks for classification into six classes. Finally, the `magic` dataset contains 19,020 instances with 10 attributes generated by a Monte Carlo programme to simulate the registration of high energy gamma particles, i.e. the task is to classify the instances

into two classes, either it is a gamma particle or a background noise.

B. Evaluation Metrics

As can be seen from the descriptions above, we have chosen a diverse set of problems to test the proposed method. To be able to evaluate the performance of the algorithm, we need a suitable scoring function. We have selected the quadratic weighted κ, which expresses the agreement between two raters. Its values range from -1 (complete disagreement) to +1 (complete agreement), a constant rater always receives a score of 0. The quadratic weighting means, that larger disagreements (i.e. predicting 1 instead of 9 for the `winequality` dataset) is quadratically more penalized than smaller disagreements. The quadratic weighted κ is especially suitable for the evaluations of models in tasks, where the order of classes actually has a reason (like in the prediction of quality of wine). For tasks with only two classes (`magic` and `wilt`) quadratic weighted κ is equivalent to unweighted κ. Even in this cases, the κ score is more informative than accuracy, especially in cases with imbalanced classes.

We used the κ for all datasets except the `ml-prove` one. The order of classes does not have any meaning in this case, thus we used the accuracy of the model instead of the κ statistic. Another reason to use the accuracy instead of κ is that the original paper about the `ml-prove` dataset [14] also uses accuracy and using the same metric allows us to compare the results more directly.

C. Machine Learning Methods

The evolved workflow contains several types of machine learning methods, these can be roughly divided into four groups: *splitters*, *mergers*, *preprocessors*, and *classifiers*.

The splitters divide the dataflow into several branches, we currently use two types of splitters: copy and k-means clustering. The copy splitter obviously copies the same data into several branches, while the k-means splitter divides the data by means of k-means clustering and send each cluster to a different branch. To improve the consistency of the k-means clustering, the clusters are re-numbered in such a way that clusters containing instances with lower ID have lower number than cluster with instances with higher ID (only the lowest instance ID in each cluster is considered). The number of clusters k is given by the structure of the DAG representing the workflow – it is the number of edges from the k-means node.

Each splitter is associated with a merger to bring the data back together. Data split by the k-means splitter can be easily combined by a simple union of the outputs, as each instance gets only one label between the split and the merge. Data split by the copy splitter, on the other hand, obtain a number of labels each, and are combined using a voting merger, each instance obtains the most common label from those it was assigned by the methods after the split. From a technical point of view, both of these mergers are implemented using the same algorithm – each instance obtains the most common label from those computed before the merger. Instances split by the k-means splitter are a special case and have only one label.

TABLE I. THE POSSIBLE VALUES OF PARAMETERS USED IN THIS WORK. METHODS, WHICH ARE NOT MENTIONED IN THIS TABLE DO NOT HAVE ANY TUNABLE PARAMETERS. THE NAMES OF THE PARAMETERS CORRESPOND TO THE NAMES IN THE SCIKIT-LEARN LIBRARY. N DENOTES THE NUMBER OF FEATURES.

Support Vector Classifier	
C	{0.1, 0.5, 1, 2, 5, 10, 15}
gamma	{0.0, 0.0001, 0.001, 0.01, 0.1, 0.5}
tol	{0.0001, 0.001, 0.01}
Logistic Regression	
C	{0.1, 0.5, 1, 2, 5, 10, 15}
penalty	{l1, l2}
tol	{0.0001, 0.001, 0.01}
Decision Tree Classifier	
criterion	{gini, entropy}
max_features	{0.05, 0.1, 0.25, 0.5, 0.75, 1}
max_depth	{1, 2, 5, 10, 15, 25, 50, 100}
min_samples_split	{1, 2, 5, 10, 20}
min_samples_leaf	{1, 2, 5, 10, 20}
PCA	
whiten	{true, false}
n_components	$\{\lfloor k \cdot N \rfloor \mid k \in \{0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1\}$
kBest	
k	$\{\lfloor k \cdot N \rfloor \mid k \in \{0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1\}$

Apart from the splitter and mergers which mostly control the data flow in the ensemble, there are also data processing nodes – the preprocessors process the data before they are fed into the classifiers. In this work we consider only two types of preprocessing – the PCA analysis and the k-best selector, which selects the k features most correlated with the target class.

Finally, there are the classifier nodes, which perform the actual classification based on the preprocessed data. In this work, we use four different classifiers – the decision tree, logistic regression, gaussian naive Bayes, and support vector classifier. All of these classifiers (as well as other methods mentioned above) are implemented using the scikit-learn package for Python. Each of the classifiers has several options, which are also set by the genetic programming. In order to make the optimization easier each parameter can have only values from a pre-defined set. The sets of values used in this work are presented in Table I.

D. Evolution Settings

In order to find the optimized ensembles, the genetic programming algorithm described in the previous section was run for a maximum of 128 generations with the population of 256 individuals. The fitness function was evaluated on a cluster with 64 cores, and each evaluation consisted of 5-fold crossvalidation. The average score (either κ or accuracy, depending on the dataset) was used as the fitness which is maximized. As some of the machine learning algorithms are not deterministic, the fitness was re-evaluated in each generation for all individuals, even those which did not change. This makes the algorithm more robust and reduces the factor of luck in the process.

Apart from the limit of generations, we also used the limit for maximum runtime. This limit did not affect the experiments with the smaller datasets (`winequality` and `wilt`), which ended after approx. 2 hours, but it was the stopping criterion actually used by the two larger datasets. The maximum runtime was set to 10 hours for the `magic` dataset and to 12 hours for the `ml-prove` dataset.

The algorithm uses tournament selection with the probability of 0.8 to select the better of the two individuals and a weak elitism (the best individual in the population is guaranteed to survive to the next generation). The crossover is realized by a simple same-type-subtree swap between two individuals with probability of 0.3 and with the maximum size of the resulted tree limited by 50 nodes. The mutation selects a node and with probability 0.3 generates a new subtree in this node with the same size as the original subtree. The maximum size of the mutated subtree is set to 10 nodes. As for the tuning of the parameters of the various machine learning methods, there is a mutation, which with probability 0.8 changes the current value to a neighboring value in the set of possible values (when ordered by the values), and with probability 0.2 changes the value to a value in the distance of two from the current value. For example, if the set of possible values is $\{0.01, 0.05, 0.1, 0.2, 0.5, 1\}$ and the current value is 0.1, the mutation with probability 0.8 changes the value to 0.05 or 0.2 and with probability 0.2 to either 0.01 or 0.5. There is also a copy operator which copies a given individual. This operator is run with the probability of 0.1.

During the initialization of the algorithm, trees of random size (maximum depth 15) are randomly generated.

Most of the parameters mentioned above were set based on our previous experience or some preliminary tuning, as the whole evolutionary process is too time consuming to make any serious parameter tuning.

E. Results

We present the results of the experiments in two parts. In the first part, we discuss the results for the two smaller dataset (`winequality` and `wilt`), and leave the two larger dataset for the other part. There are two reasons for this distinction. One of them is that we experimented with the two smaller dataset earlier and we compare the new results to the old ones, and the other reason is, that the two larger datasets may be considered a more typical scenario for the approach presented here and there are some interesting challenges for the algorithm which are revealed only with this larger datasets.

The results for the two small datasets are presented in Table II. The table also contains the results of each of the machine learning method applied directly on the data with default and with tuned hyper-parameters. The tuning was done using a grid search.

The "systematic" line refers to an earlier experiment, where instead of using genetic programming we generated the possible ensembles using a systematic, A*-based approach with fixed values of all hyper-parameters – either default values or those obtained by hyper-parameter tuning of the single models. The GP line shows the result obtained by the approach described in this paper and it is the quadratically weighted κ obtained by the best individual in the last generation and averaged over five runs.

The difference between the systematic generating of workflows and the genetic programming approach seems rather small, however, the GP technique is much more effective in the searching. The systematic approach was used to generate 65,536 different workflows, while the GP generated only half of that. Moreover, GP is able to react to the performance of various techniques and thus uses and evaluates mostly those with better performance. In the case of these two datasets this leads to significant savings of computational time, as the slowest method (SVC) is only rarely used, and thus GP obtained the results several times faster than the systematic approach. This is also demonstrated in Figure 4, which shows that the average evaluation time of an ensemble sometimes drops despite the fact, that the size of the ensembles tend to grow during the evolution.

TABLE II. RESULTS FOR THE `WILT` AND `WINEQUALITY` DATASETS, THE QUADRATIC WEIGHTED κ FOR THE DESCRIBED METHODS. FOR GP THE NUMBERS ARE THE AVERAGE OF FIVE INDEPENDENT RUNS.

dataset	winequality		wilt	
params	default	tuned	default	tuned
SVC	0.1783	0.3359	0.0143	0.8427
LR	0.3526	0.3812	0.3158	0.6341
GNB	0.4202	0.4202	0.2916	0.2917
DT	0.3465	0.4283	0.7740	0.8229
systematic	0.4731	0.4756	0.8471	0.8668
GP	**0.4792**		**0.8702**	

The precise speed-up is hard to evaluate, as we run both the experiments in different environments, however, the evaluation of the systematically generated workflows took around 4 hours using more than 200 CPU cores (a mix of Intel i7 at 2.66GHz and Intel Core2 Quad at 2.83GHz, quadcore CPUs in both cases), while the GP took less the two hours with 64 cores (Intel Xeon at 2.4GHz, four 16-core machines).

For the `winequality` dataset, the difference between the systematic and GP approaches is rather small and on average the GP found a solution similar to the one found by the systematic generating. This is probably caused by the fact that the best systematic solution is rather simple and contains only the GNB which has no parameters, it is easy to find such a solution by the evolution. However, in two out of the five runs, GP found a better solution, the best of them had the value of $\kappa = 0.4898$.

The GP approach in four out of five runs found a better solution than the systematic one for the `wilt` dataset, thus proving to be superior in this case. The best solution had the $\kappa = 0.8756$.

The evolution of ensembles for the two larger datasets was again able to outperform the simple models. We did not try to evaluate the systematically generated ensembles in this case, as the whole process is much more time consuming. In this case, the stopping criterion of the GP was set to 10 hours for the `magic` dataset and to 12 hours for the `ml-prove` dataset. This led to approximately 30 generations in the former case, and to around 70 generations in the later case. With this larger datasets the training time of the various models becomes important. The SVCs are the slowest of the models and in many cases there was a few individuals in the population which contained a lot of them, thus slowing down the evolution considerably (and making the use of computational resources ineffective as many of the cores were idle waiting for the few slow individuals). It may be interesting to study how to set the number of individuals and the number of cores in order to use the resources more effectively.

Apart from the numerical results, we also provide two types of graphs, which show the behavior of the algorithm. In Figure

3 we show how the score of the best and average individual in the population changes with the number of generations. It shows, that it is quite easy to outperform the simple methods with an ensemble, as almost from the beginning there is are some individuals which are better. However, overperforming the systematically generated ensembles is much more complicated.

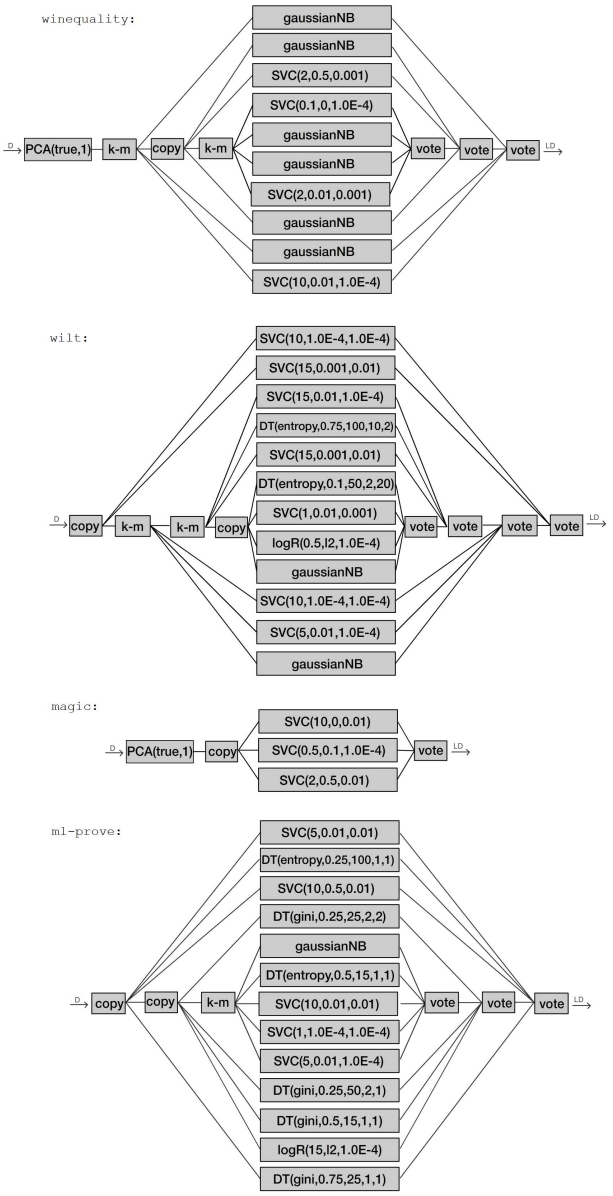

Fig. 2. The best ensembles for the `winequality`, `wilt`, `magic` and `ml-prove` datasets.

The resulting ensembles are depicted in Figure 2. The parameters of the models are written in the same order as they are in Table I, however, for the k-best and PCA, the pictures contain the fraction of the input features which should be in the output instead of the number as this value makes more sense during the evolution. The pictures show that in most cases the GP found interesting ensembles which would be hard to find manually.

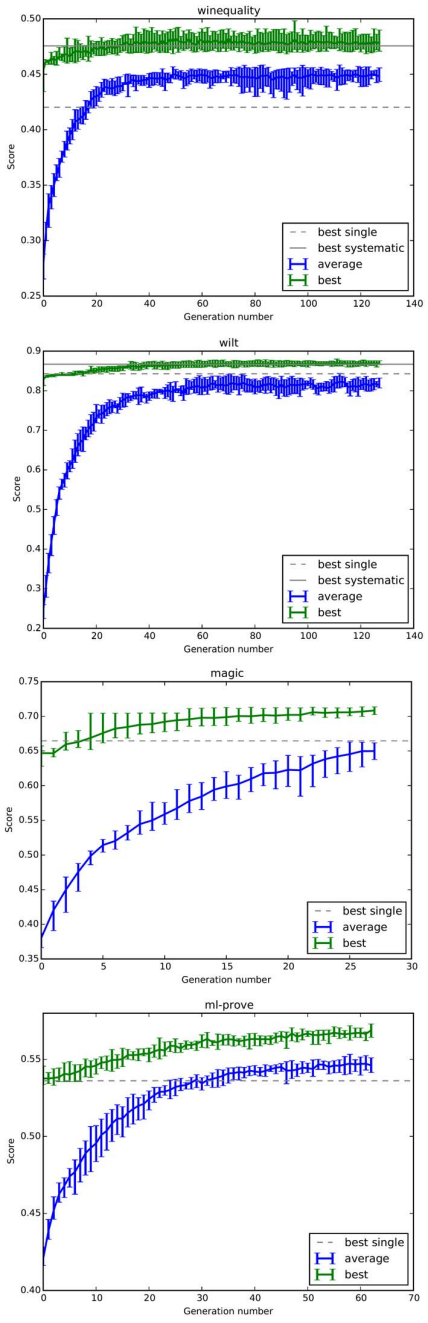

Fig. 3. The aggregated values (over 5 runs) of average and best score in the population. The error bars show the minimum and maximum values, the line is the average.

V. Conclusion

We have presented an approach for the creation of ensembles based on typed genetic programming. The approach was evaluated on four datasets coming from different areas and shows promising results for all of them. The evolved ensembles are clearly superior to the use of simple methods, and they

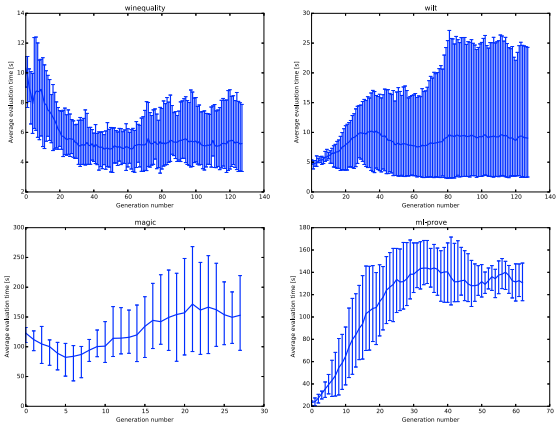

Fig. 4. The aggregated values (over 5 runs) of the size of the average evaluation time of the ensembles in the population. The error bars show the minimum and maximum values, the line is the average.

TABLE III. RESULTS FOR THE ML-PROVE AND MAGIC DATASETS. QUADRATIC WEIGHTED κ FOR THE MAGIC DATASET AND ACCURACY FOR THE ML-PROVE DATASET. FOR GP THE NUMBERS ARE THE AVERAGE OF 5 INDEPENDENT RUNS.

dataset	ml-prove	magic
SVC	0.5361	0.6147
LR	0.4574	0.5178
GNB	0.1696	0.3289
DT	0.5190	0.6647
GP	**0.5689**	**0.7084**

are also better then systematically created ensembles. Another advantage of the GP-based approach is that it evolves the shape and the parameters of the methods at once and thus saving computational resources.

Our tree generating method provides a novel approach to effective population initialization and mutation for typed genetic programming with parametric polymorphism. We have also demonstrated a technique of lifting natural numbers into type system which in turn opens doors for more involved type level programming in the fashion of logic programming.

There are many possible areas for future extension of this approach, it would be interesting to add more ensembling techniques than voting. It is also important to make sure, that the evolution is more effective, killing the long-computing tasks may be an interesting option, which would improve the runtime of the algorithm considerably, and could also lead to faster models.

ACKNOWLEDGMENT

Martin Pilát and Roman Neruda have been supported by the Czech Science Foundation project no. P103-15-19877S. Tomáš Křen has been supported by the Grant Agency of the Charles University project no. 187115 and by SVV project number 260 224. This work was supported by the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), funded by the European Regional Development Fund and the national budget of the Czech Republic via the Research and Development for Innovations Operational Programme, as well as Czech Ministry of Education, Youth and Sports via the project Large Research, Development and Innovations Infrastructures (LM2011033).

REFERENCES

[1] T. Křen, M. Pilát, K. Pešková, and R. Neruda, "Generating workflow graphs using typed genetic programming," in *Proceedings of the MetaSel 2015*, 2015, in print.

[2] P. Panov, S. Dzeroski, and L. Soldatova, "Ontodm: An ontology of data mining," in *Data Mining Workshops, 2008. ICDMW '08. IEEE International Conference on*, Dec 2008, pp. 752–760.

[3] C. Diamantini, D. Potena, and E. Storti, "Ontology-driven kdd process composition," in *Advances in Intelligent Data Analysis VIII*, ser. Lecture Notes in Computer Science, N. Adams, C. Robardet, A. Siebes, and J.-F. Boulicaut, Eds. Springer Berlin Heidelberg, 2009, vol. 5772, pp. 285–296. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03915-7_25

[4] O. Kazík and R. Neruda, "Data mining process optimization in computational multi-agent systems," in *Agents and Data Mining Interaction - 10th International Workshop, ADMI 2014, Paris, France, May 5, 2014*, ser. LNAI 9145. Springer, 2015, p. in print.

[5] G. Folino and F. Pisani, "Combining ensemble of classifiers by using genetic programming for cyber security applications," in *Applications of Evolutionary Computation*, ser. Lecture Notes in Computer Science, A. M. Mora and G. Squillero, Eds. Springer International Publishing, 2015, vol. 9028, pp. 54–66. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16549-3_5

[6] F. Briggs and M. O'Neill, "Functional genetic programming and exhaustive program search with combinator expressions," *International Journal of Knowledge-Based and Intelligent Engineering Systems*, vol. 12, no. 1, pp. 47–68, 2008.

[7] F. Binard and A. Felty, "Genetic programming with polymorphic types and higher-order functions," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*. ACM, 2008, pp. 1187–1194.

[8] T. Yu, "Hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction," *Genetic Programming and Evolvable Machines*, vol. 2, no. 4, pp. 345–380, 2001.

[9] R. Milner, "A theory of type polymorphism in programming," *Journal of computer and system sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[10] T. Křen and R. Neruda, "A dynamic programming approach to individual initialization in genetic programming," in *Proceedings of the IEEE SMC 2015 conference*, 2015, in print.

[11] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[12] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis, "Modeling wine preferences by data mining from physicochemical properties," *Decision Support Systems*, vol. 47, no. 4, pp. 547 – 553, 2009, smart Business Networks: Concepts and Empirical Evidence. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167923609001377

[13] B. A. Johnson, R. Tateishi, and N. T. Hoan, "A hybrid pansharpening approach and multiscale object-based image analysis for mapping diseased pine and oak trees," *Int. J. Remote Sens.*, vol. 34, no. 20, pp. 6969–6982, Oct. 2013. [Online]. Available: http://dx.doi.org/10.1080/01431161.2013.810825

[14] J. Bridge, S. Holden, and L. Paulson, "Machine learning for first-order theorem proving," *Journal of Automated Reasoning*, vol. 53, no. 2, pp. 141–172, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10817-014-9301-5

[15] R. Bock, A. Chilingarian, M. Gaug, F. Hakl, T. Hengstebeck, M. Jiřina, J. Klaschka, E. Kotrč, P. Savický, S. Towers, A. Vaiciulis, and W. Wittek, "Methods for multidimensional event classification: a case study using images from a cherenkov gamma-ray telescope," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 516, no. 2–3, pp. 511 – 528, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168900203025051