Semillero Automated Software Configuration - Final installment

Jose Acevedo

November 2020

1 AI Planning

Within the field of Artificial Intelligence, AI Planning focuses on exploring techniques that can solve planning problems given an initial state and a goal state that is achieved through the execution of a series of actions, which will constitute a plan [2, p.61].

Planning problems can be attacked in two major ways:

- Domain-specific approaches in which the planner has access to domain knowledge and heuristics. This approach has the problem of not transferring very well to another application domain [2, p.61].
- Domain-independent approaches that try to solve the general problem without using domain knowledge. This approach's aim is to be able to solve a wide range of problems and the larger part of AI research focuses on this approach [2, p.61].

There are potentially infinite search spaces, or possible ways that a planner must choose from in order to reach a desired goal state thus, performance measures that a planner can optimize need to be established in order to obtain solutions for a given initial state.

1.1 Classical Planning

Planning researchers have settled on a factored representation—one in which a state of the world is represented by a collection of variables [4, p.367]. These representations assume that this world that is being mentioned is closed, only actions a change its state, and it is deterministic; the same set of actions will always yield the same result. In this subsection we'll elaborate on some of the formalism used here in order to build up to HTN.

A *classical planning domain* is defined as $\Sigma = (S, A, \gamma)$ [1, p.25]:

- S is finite set of *states* in which the system may be. A single state is written as s.

- A is a finite set of *actions* that the actor may perform. A single action is written as a.
- $\gamma : S \times A \to S$ is a *state-transition function* that, if $\gamma(s, a)$ is defined, then that means that a is *applicable* in s, with $\gamma(s, a)$ being the predicted outcome.

1.1.1 Objects and State variables

In said *classical planning domain*, each state $s \in S$ is a description of the properties of various objects in the planner's environment. Properties can be *rigid* if they are the same in every state in S, and *varying* if it changes between states s. In order to represent objects, we will use three sets B, R, and X, which have to be finite [1, p.28]:

- B is a set of names for all the objects, plus any constants that may be needed to represent properties of those objects.
- To represent Σ's rigid properties, we will use a set R of *rigid relations*. Each $r \in R$ will be an n-ary relation over B
- To represent Σ's varying properties, we will use a set X of *state variables* whose values depend solely on the state s.

1.1.2 Actions and action templates

A literal is an expression that has either of the following forms [1, p.32]:

$$rel(z_1, ..., z_n) \qquad or \qquad sv(z_1, ..., z_n)$$

A literal can be *ground* if it contains no variables, and *unground* otherwise.

An *action template* is a tuple α=(head(α), pre(α), eff(α)), which each element being defined as follows [1, p.33]:

- head(α) is a syntactic expression of the form

  $$act(z_1, z_2, ..., z_k),$$

  where *act* is the *action name*, and $z_1, z_2, ..., z_n$ are its *parameters*, which include all the variables that appear in pre(α) and eff(α), with each parameter z_i having Range(z_i) $\subseteq B$.
- pre(α) $= p1, ..., p_m$ is a set of *preconditions*, each being a literal.
- eff(α) $= e_1, ..., e_n$ is a set of *effects*, each being an expression of the form

  $$sv(z_1, ..., z_n) \leftarrow t_0$$

  where the literal is the effect's target, and t_0 is the value to be assigned.

A *state-variable or classical planning action* is simply a ground instance a of an action template α. An action a is *applicable* in a state s if it satisfies $\text{pre}(\alpha)$, and the predicted outcome of doing so is the result of the *state-transition function* $\gamma(s, a)$ [1, p.34].

1.1.3 Defining the planning problem

We can now define a plan as a finite sequence of actions

$$\pi = \langle a_1, a_2, ..., a_n \rangle$$

with $|\pi| = n$ being the plan's length.

We can now define what it means for a plan to be applicable in a state s_0. A plan $\pi = \langle a_1, ..., a_n \rangle$ is applicable in a state $s_0$if there are states $s_1, ..., s_n$ where $s_i = \gamma(s_{i-1}, a_i)$.

A state-variable planning problem or classical planning problem is a $P(\Sigma, s_0, g)$ with Σ being the planning domain we previously defined, s_0 the initial state and a goal g as a set of ground literals which a planner will attempt to find a plan for such that the state $\gamma(s_0, \pi)$ satisfies g (which means that after such plan π is applied to the inital state s_0, the g set of ground literals is produced).

1.2 Hierarchical Task Networks

We can see Hierarchichal Planning with Hierarchical Task Networks as an extension of Classical Planning, however, there's no conception of a *goal state*, but rather of an initial state s_0 and an initial *task network* with variables that are grounded by the planner.

In order to achieve this, a logic language L needs to be defined; with elements such as variable names, constant names, predicate names, quantifiers and connectors to build formulas, all of the components that will be discussed are defined in terms of said L.

Hierarchical planning also expands on the concept of actions from classical planning by adding the following concepts:

- *Operators* are tuples $\langle name, pre, post \rangle$ which represent the name of the operation, preconditions and postconditions respectively [3].
- *Methods* are tuples $\langle name, task, pre, T \rangle$ which represent the name of the method, its non-primitive task, a logical precondition and a task network respectively. The previously mentioned task network decomposes the method iteratively [3].
- *Tasks* can either be primitive (described by an operator) or complex, in which case are described by methods that can be grounded multiple times until a simple task is achieved. A single task is written as

  $$t(v_0, ..., v_n)$$

  where t is the task name, and $v_0, ..., v_n$ a list of parameters, which are a variables or constants from L [3].

We can now define a simple task network planning problem as a quadruple $\langle cans_0, T_0, O, M \rangle$ where O and M are sets of operator and methods, respectively. [3]

Figure 1: Tasks diagram in HTN.

With this clear, we can define our planning problem given an initial state s_0 and a task network T_0. The problem is to obtain a plan π that whose members are ground operations, also called *actions*, applicable in s_0. More formally, the planning problem is expressed as a tuple

2 HASCO

HASCO is a domain-independent algorithm whose main goal is to configure and parametrize software components via the notion of provided, required interfaces and their satisfaction. To do this, HASCO goes through 3 levels or layers, each with their own logic. It first reduces the problem to a HTN planning problem with it's own language L, then it is reduced again into a Graph-search problem in which the search phase is executed until it yields a solution, then this solution is decoded twice until it reaches the Software Configuration level.

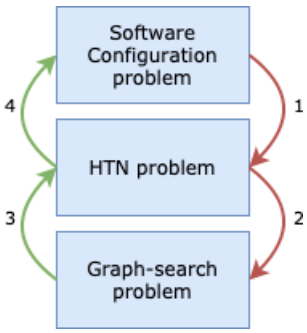

Figure 2: HASCO reduction and decoding steps

The key advantages of this approach is that logic from every level can be changed without needing to modify the logic of the other layers, only the transition steps (reduction and decoding). For instance, the way HASCO does the graph search could be changed without having to rewrite logic from other parts. This separation, also gives us access to a wide variety of already established search algorithms that, thanks to this reductive approach, can be leveraged to

solve the initial software configuration problem. Not attempting to solve the initial problem directly gives us flexibility down the line and helps us decouple logic.

2.1 Encoding Software Configuration problems as HTN planning problems

A software component consists of a name, its required interfaces, provided interfaces and its parameters. HASCO takes a group of said components, an initial *required interface* and a composition evaluate function. For every component, HASCO generates the appropriate methods and operation that allow for the satisfaction of interfaces recursively and automatic refinement of component parameters; we can see this as a way of expressing what *can* or *can not* be done for the next level to use.

It is also important to understand the difference between components and component instances. Components are obtained from the first level of HASCO during the 1st step and contain information such as its name, provided interfaces, required interfaces and parameters. While component instances are what we get in the 4th step during the last decoding phase and contain ground values for the parameters and already connected different component instances with each other through the satisfaction of required interfaces.

As a last clarification, characters enclosed in chevrons or angled brackets are replaced by names. For example `<i>` will be replaced by the name of an interface when a task/operation/method is generated by HASCO:

2.1.1 Tasks

In this part we'll introduce the tasks that HASCO creates in the HTN level. Keep in mind that a task is just a name and a set of parameters, so they have to be delegated to either a method (complex) with a task network of its own, each of those tasks is delegated until it reaches a primitive task that points to an operation or a operation (primitive).

Complex tasks

- `tResolve<i>(c1,c2)` This task resolves the interface `<i>` of a *component instance* c1 with the *component instance* c2.
- `tRefineParamsOf<c>(c, p1, ..., pm)` This task begins the parameter refinement for a *component instance* c, these parameters are declared from the software configuration level.
- `tRefineParam<p>Of<c>(ci, pc)` This task begins the refinement of the parameter pc that belongs to a *component instance* ci.

Primitive tasks

- `satisfy<i>With<c>(c1, c2, p1, ..., pm, r1, ..., rn)` This task declares in the state that the interface `<i>` of *component instance* c1 is satisfied by a *component instance* c2 of component type `<c>`.

- `redefValue(container, previousValue, newValue)` Changes the value of `container` from `previousValue` to `newValue`.
- `declareClosed(container)` Declares in the state that `container` will not be changed anymore (the refinement process is complete).

2.1.2 Operations

There's only one type of operation, `satisfy<i>With<c>(c1, c2, p1,..., pm, r1,..., rn)`. It is generated for every *provided interface* of every component defined in the software configuration layer. Its purpose is to express that the interface `<i>` is satisfied by the component `<c>`; keep in mind that these operations are generated whether or not the final solution uses them or not. Let's remember that in this step HASCO is defining what the planner can do through tasks, methods and operations. It is not planning just yet.

2.1.3 Methods

We could see methods as implementations of tasks, grouping functionality through *task networks*.

- `tResolve<i>With<c>(c1; c2, p1, ..., pm, r1, ..., rn)` This method triggers the satisfaction of the interface `<i>` with the component `<c>`, then triggers `tResolve<i>` for each of the interfaces of the component instance c2 (this method is basically calling itself until there are no more required interfaces to be satisfied). Lastly it triggers the refinement of parameters for the component instance c2 of component type `<c>`.
- `ignoreParamRefinementFor<p>Of<c>(object, container, curval)` This method stops the parameter refinement for parameter `container` of parameter name `<p>`.
- `refineParam<p>Of<c>(object, container, curval)` This method represents one step of the parameter refinement process for the parameter `container` of name `<p>`, it redefines the value of said parameter once.
- `tRefineParamsOf<c>(c2, p1, .., pm)` In this method we trigger a parameter refinement step for each parameter in the argument list, then at the end call the task `tRefineParamsOf<c>(c2, p1, .., pm)` which could call this method again and repeat the aforementioned process or trigger the next method that closes the refinement of the parameters.
- `closeRefinementOfParamsOf<c>(c2, p1, ..., pm)` This method stops parameter refinement by having an empty task network, contrary to the previous method with the same task name`tRefineParamsOf<c>(c2, p1, .., pm)` which refines said parameters.

Recently, HASCO was expanded to support components that require multiple interfaces, this could mean a group of *required interfaces* need to be satisfies

from 0 (optional case) to n times. To support this type of problems, new methods are generated to replace the functionality of `tResolve<i>With<c>(c1; c2, p1, ..., pm, r1, ..., rn)` originally explained above which only works for non-list interface problems; also a new type of operation is created. This allows the graph search layer to generate desired plans for this new type of problem.

One final thing to keep in mind is that required interfaces in this type of problems have a *minimum* and *maximum* number of times they can be satisfied for the same component who requires them:

2.1.4 New methods for list interfaces

- `resolve<i>(c1; c2_1, ..., c2_<max(I)>)` Begins the resolution of the required interface `<i>` that belongs to the component instance c1, triggering a satisfaction step that looks for a *single* component. This satisfaction step is run `min(I)` times. Then a similar version of the previous single component resolution step is executed, but in this case, the resolution of the interface is optional, this is executed `max(I)-min(I)+1` times.
- `resolve<i>With<c>(c1, c2; p1, ..., pm, r1, ..., rn)` In this method is where a required interface `<i>` is declared satisfied and the resolution of the component c2 of component type `<c>`'s required interfaces is triggered. Finally, its task network calls the task to refine c2's parameters.
- `doResolve<i>(c1, c2)` This method's task name is `tResolveSingleOptional<i>(c1, c2)`, and represents the case in which *optional* takes the case of present (an optional interface will be present in the composition), therefore, delegating the resolution of the interface `<i>` to the method above.
- `doNotResolve<i>(c1, c2)` This method's task name is also `tResolveSingleOptional<i>(c1, c2)`, but it represents the case where *optional* means not present for the nth time the required interface `<i>` of c1.

2.1.5 New operation for list interfaces

The new operator is `omitResolution(c1, i, c2)`; it ensures that after triggering `doNotResolve<i>(c1, c2)`, the $n+1$th required interface will *not* be solved. For example, if a required interface has a `min= 2` and a `max=5` and the resolution for the interface triggered `doNotResolve<i>(c1, c2)` for the 3rd time, the 4th and the 5th time's resolutions will be omitted.

2.2 Decoding the HTN solution to Software Configuration solution

After the graph search layer finds a solution, it is then decoded into a plan in the HTN level. Now we need to understand how this plan is translated into a

Software Configuration composition or component instance which is, ultimately, the solution to the software configuration problem we had initially.

The solution in the HTN layer is a final state s_f which consists of a set of literals. The planner might return us a plan $\pi = \langle a_1, ..., a_n \rangle$, but remember that $\gamma(s_0, \pi)$ yields the final state of our HTN problem. Now a strategy that could be used to create component instances is to retrieve the literals in the final state s_f that match the patterns that were established in the language L and compose the component instances that solve the original problem.

3 Applying list interfaces in HASCO to ensemble learning

Software components in HASCO can declare required interfaces and their cardinalities, which in turn can be satisfied by multiple other components; this makes a perfect use case for machine learning ensembles. Now it's time to test the performance of HASCO for ensemble learning against an AutoML tool, ML-Plan [3], which is very competitive and supports pipelines, preprocessors and classifiers, and also evaluates possible candidates at runtime and finds solutions using HTN. However, ML-Plan doesn't yet support ensembles, and the purpose of the following experiments is to find out how well HASCO works for ensemble problems.

3.1 Setting up the experiment

In order to measure each algorithm's performance, we executed each one for an hour 5 times on 5 datasets and saved the error rates from their best solutions and the classifiers chosen in every run. These tests were executed in two VMs, each with 4 cores and 16, 32 and 128 GB of memory (we scaled memory depending on the dataset), and keeping conditions fair by maintaining the same hardware specs and JVM settings for each algorithm execution on the same dataset.

3.1.1 Search space

We configured HASCO's search space to consist of an initial ensemble component that requires components that provide the AbstractClassifier interface, which in this case are Weka classifiers; said interface has a cardinality of minimum 3 and maximum 11, which allows the generation of ensemble candidates of a variety of sizes.

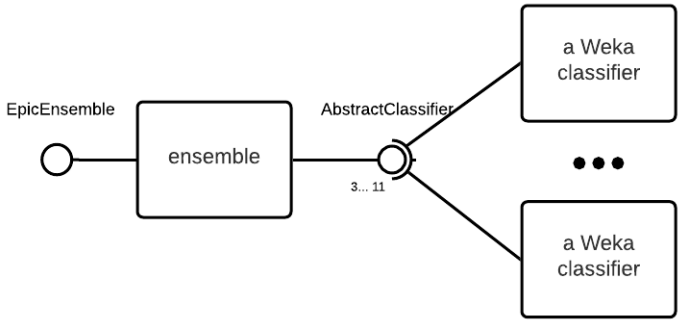

Figure 3: HASCO's search space for experimentation.

3.1.2 Datasets

We used the following 5 datasets available at `https://www.openml.org` for our experiments.

dataset	# of instances	# of features	# of classes
dexter	600	20001	2
madelon	2600	501	2
dorothea	1150	100001	2
amazon-commerce-reviews	1500	10001	50
har	10299	562	6

3.2 Results

3.2.1 Dexter dataset

algorithm	repetition	error rate	solution
MLPlan	1	0.0833	SMO
MLPlan	2	0.0500	SMO
MLPlan	3	0.0444	SMO
MLPlan	4	0.0555	SMO
MLPlan	5	0.0500	SMO
HASCO	1	0.0833	RandomSubspace, VotedPerceptron, J48
HASCO	2	0.1111	RandomSubspace, VotedPerceptron, J48
HASCO	3	0.0833	RandomSubspace, VotedPerceptron, J48
HASCO	4	0.0722	RandomSubspace, VotedPerceptron, J48
HASCO	5	0.0833	RandomSubspace, VotedPerceptron, J48

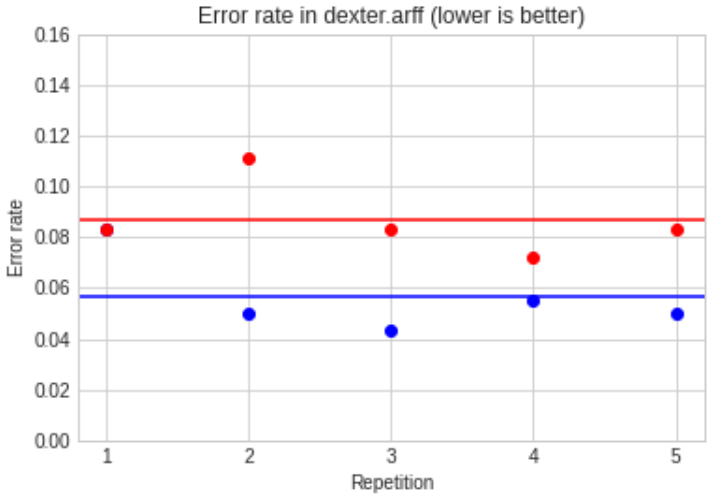

Figure 4: Results for dexter.arff along with mean. ML-Plan in blue and HASCO in red.

3.2.2 Madelon dataset

algorithm	repetition	error rate	solution
MLPlan	1	0.2141	Bagging
MLPlan	2	0.2795	REPTree
MLPlan	3	0.2795	REPTree
MLPlan	4	0.2154	JRip
MLPlan	5	0.3166	REPTree
HASCO	1	0.2564	RandomSubspace, VotedPerceptron, J48
HASCO	2	0.2769	RandomSubspace, VotedPerceptron, J48
HASCO	3	0.2717	RandomSubspace, VotedPerceptron, J48
HASCO	4	0.2513	RandomSubspace, VotedPerceptron, J48
HASCO	5	0.2666	RandomSubspace, VotedPerceptron, J48

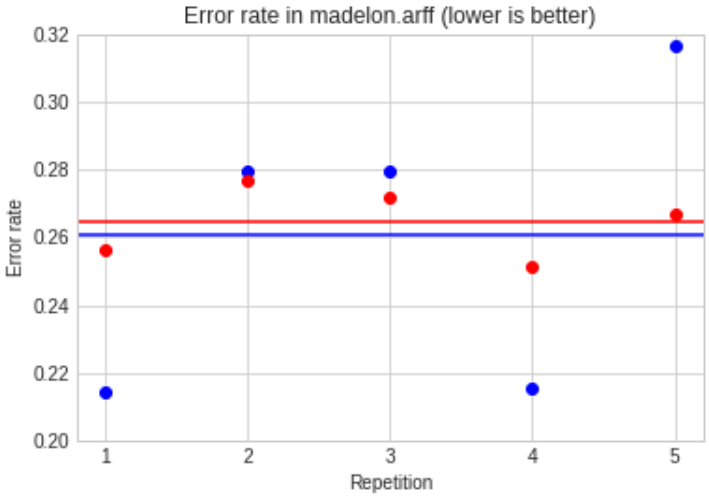

Figure 5: Results for madelon.arff along with mean. ML-Plan in blue and HASCO in red.

3.2.3 Dorothea dataset

HASCO results are incomplete because it kept crashing on this specific dataset even after multiple re-runs. It may have to do with the fact that Dorothea has 100001 features, which is the most out of the 5 datasets we used in our experiments, and our implementation of the ensemble layer is sub-optimal.

algorithm	repetition	error rate	solution
MLPlan	1	0.0725	OneR
MLPlan	2	0.0725	VotedPerceptron
MLPlan	3	0.0696	OneR
MLPlan	4	0.0580	VotedPerceptron
MLPlan	5	0.0609	OneR
HASCO	1	0.0638	RandomSubspace, VotedPerceptron, J48
HASCO	2	0.0609	RandomSubspace, VotedPerceptron, J48

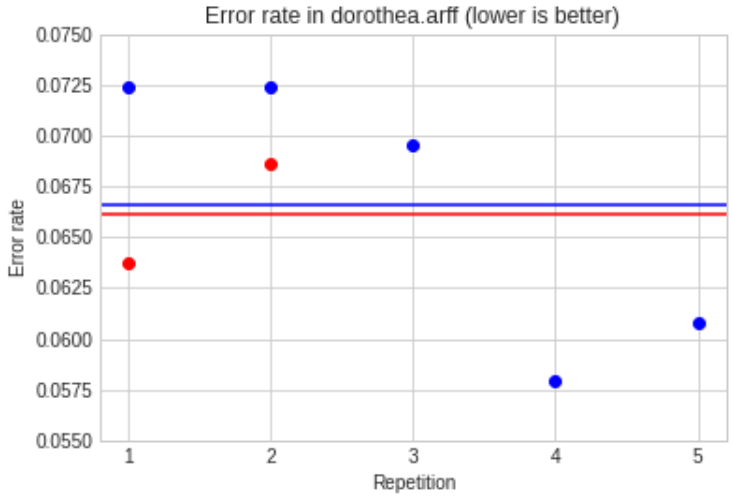

Figure 6: Results for dorothea.arff along with mean. ML-Plan in blue and HASCO in red.

3.2.4 Amazon Commerce Reviews dataset

algorithm	repetition	error rate	solution
MLPlan	1	0.2977	NaiveBayesMultinomial
MLPlan	2	0.4577	NaiveBayes
MLPlan	3	0.3177	NaiveBayesMultinomial
MLPlan	4	0.5711	J48
MLPlan	5	0.5644	J48
HASCO	1	0.5200	RandomSubspace, J48
HASCO	2	0.4711	RandomSubspace, J48
HASCO	3	0.4888	RandomSubspace, J48
HASCO	4	0.5311	RandomSubspace, J48
HASCO	5	0.4911	RandomSubspace, J48

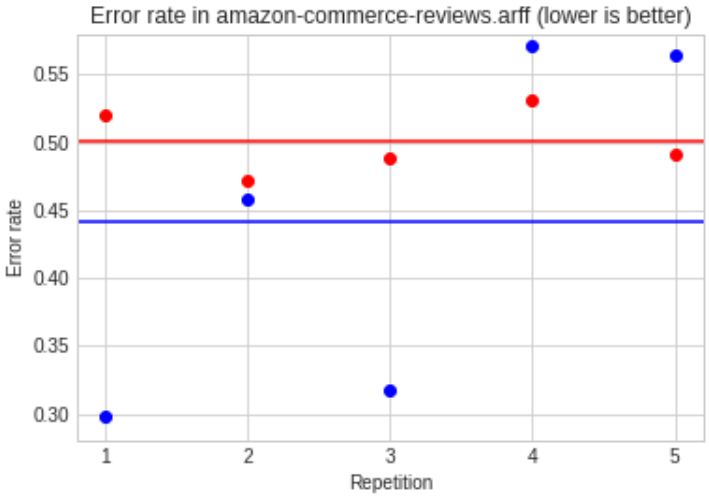

Figure 7: Results for amazon-commerce-reviews.arff along with mean. ML-Plan in blue and HASCO in red.

3.2.5 Human Activity Recognition (HAR) dataset

algorithm	repetition	error rate	solution
MLPlan	1	0.0162	SimpleLogistic
MLPlan	2	0.0165	SimpleLogistic
MLPlan	3	0.0239	RandomForest
MLPlan	4	0.0227	RandomForest
MLPlan	5	0.0184	SimpleLogistic
HASCO	1	0.0333	RandomSubspace, J48
HASCO	2	0.0304	RandomSubspace, J48
HASCO	3	0.0382	RandomSubspace, J48
HASCO	4	0.0369	RandomSubspace, J48
HASCO	5	0.0324	RandomSubspace, J48

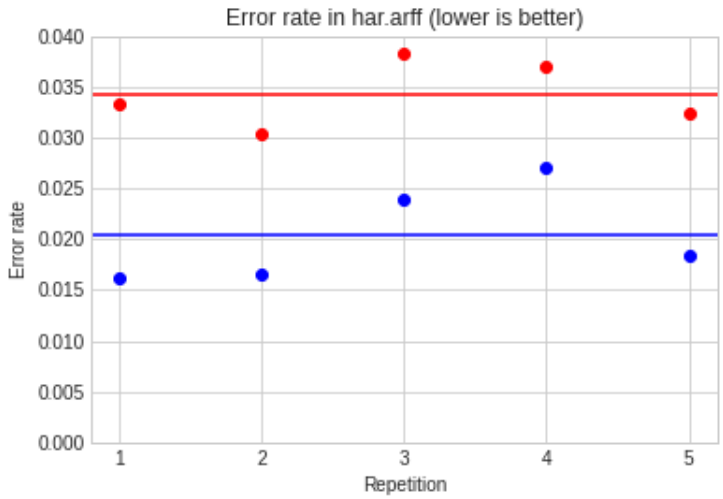

Figure 8: Results for HAR.arff along with mean. ML-Plan in blue and HASCO in red.

3.3 Results analysis

- HASCO ensembles' error rate is closer to its error mean over the 5 repetitions than MLPlan's. However, checking the logs from HASCO's execution, we see that building ensembles takes a very high amount of time, which could limit the number of solutions that HASCO can consider in a given amount of time. This paired with the fact that the classifiers chosen across different datasets and repetitions are very similar, if not the same for most runs, leads us to believe that this low variance in performance is caused by very similar solutions, which are in turn caused by a mostly unexplored search tree caused by inefficient candidate evaluation from our ensemble implementation.

- MLPlan performs better on every dataset, except for Dorothea, though we only managed to run HASCO ensembles twice on it, which isn't sufficient to make claims of superior performance. However, we're surprised that Hasco ensembles' performance isn't very far behind MLPlan's despite what we discussed in the previous observations.

- We did not see clear relationships between number of instances, number of features or number of classes from datasets and the difference between the mean error rates of both algorithms, which suggests that in our experiments, the dataset didn't significantly affect performance difference between HASCO ensembles and MLPlan.

References

[1] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

[2] James A. Hendler, Austin Tate, and Mark Drummond. Ai planning: Systems and techniques. *AI Magazine*, 11(2):61, Jun. 1990.

[3] Felix Mohr, Marcel Wever, and Eyke Hüllermeier. Ml-plan: Automated machine learning via hierarchical planning. *Machine Learning*, 107(8):1495–1515, Sep 2018.

[4] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.