# Feature-conservative portfolio SAT: Feature computation and solver-switching as a sequential decision process

Kevin Rosa, Ajay Natarajan, Luis Costa, Eugene Shao, Albert Zhai

Github: https://github.com/krosa100/Feature-conservative-portfolio-SAT

## Introduction

In many problems, it is possible to develop heuristics which perform well on certain types of problem instances, producing solvers which run reasonably quickly on realistic distributions of inputs. As different heuristics succeed under different conditions, the question of “algorithm selection” arises: how should we choose which solver to deploy on a novel problem instance?

The answer may lie in algorithm portfolios— a method for automatically performing algorithm selection by building models of each candidate algorithm’s performance. For most portfolio-based approaches, a hand-crafted set of features is computed for each instance in a dataset of problem inputs. Each candidate solver is deployed on each instance, and a regression model  is trained to predict the solver’s runtime from the computed problem features. On new instances, the solver with the lowest predicted runtime is then chosen for use.

In practice, algorithm portfolios are usually able to successfully select appropriate algorithms in various situations, enabling them to achieve winning performance in competitions such as the annual SAT Competition for Boolean satisfiability problems. However, the computation of the full set of problem features for each new input often imposes a significant overhead on the total runtime of the portfolio. In the 2008 SAT Race, feature computation time alone exceeded the runtime of the actual solvers in many cases. Thus, portfolio speed could be drastically improved by reducing time spent computing features.

In this project, we view feature computation as a sequential decision process, and propose a method for adaptively deciding when to stop computing features. Our method iterates through the feature set and at each step, using lightweight learned models, estimates the runtime improvement of the final selected solver resulting from computing the next feature. The algorithm saves time by stopping early when the estimated improvement is smaller than the estimated feature computation time. The stopping decision navigates a fundamental tradeoff between computation time and solver selection optimality -- by ignoring features, we sacrifice runtime prediction accuracy in favor of reduced overhead. We plan to target empirical evaluation of our method at SAT problems, where portfolios have demonstrated the most success.

## Background and previous work

**Feature Runtime prediction** Heavy feature computation times have led to work on feature runtime estimates— one of the extensions the developers of SATzilla made to the original (2007) version in 2009 was exactly these estimates [7]. However, this extension was merely linear regression. Eggensperger et al showed that feature runtime distributions (RTDs) could be well predicted by neural networks [6]. They use a parametric approach, where they decide which distribution best models the observed runtimes via MLE and then use a neural network to map features of the problem instance to the distribution’s parameters. Their paper showed that

neural networks can be used to jointly learn distribution parameters to predict runtime distributions (RTDs) and obtain better performance than previous approaches such as random forests. In our work, we fit a Gaussian mixture model to the generative distribution of feature runtimes and take conditional expectations for decision-making.

**Algorithm Scheduling** Portfolio solvers like SATzilla have mostly focused on algorithm selection: For a given problem instance, select the best solver. However, this can be extended to algorithm *scheduling,* where one runs several solvers on the same problem instance. Such a schedule would determine which solvers are run on the problem instance and for what amount of time. One of the first to introduce this approach were O'Mahony et al. [2], in their constraint-satisfaction portfolio solver CP-Hydra. The technique was also utilized in a SAT portfolio solver in an effort to extend SATzilla [3]. The authors produced a combination of scheduling and solver selection that significantly boosted performance with respect to SATzilla. Here, we focus on choosing a single best solver, but note that our approach can be extended to employ multiple-solver scheduling.

**Deep Learning** In more recent years, the idea of using neural feature representations to guide SAT solving has gained increasing traction. For instance, Kurin et al. showed that augmenting SAT solvers with agents trained with RL and graph neural networks can improve performance [1]. Their approach enables them to calculate a branching heuristic from data, which they show to be far more effective than previously used hand-crafted branch heuristics. Moreover, D. Selsam et al. were able to create a neural network, NeuroSAT, that could effectively guide the variable branching decisions of high-performance SAT solvers such as Glucose (20% more problems solved before timeout), MiniSat (10%) and Z3 (6%) [4]. Jaszczur et al. use the graph representations of the SAT problems similar to those used by D. Selsam et al. to enhance the performance of well-known backtracking algorithms such as DPPL and CDCL. [5]. Their results corroborate the idea that graph neural networks can be used to augment existing SAT solvers. In our project, we do not use deep learning to improve individual solver heuristics, but do investigate learning deep representations of SAT instance features to parameterize a decision policy for sequential feature selection.

## Research question

We consider a generalization of the portfolio SAT strategy where we intelligently choose which feature to compute next or which solver to run. We consider a model-based and model-free approach to the strategy and ask:

- **Feature-conservative portfolio SAT:** To what extent are these methods able to achieve improved performance over strategies which do not as generally choose which feature to compute next (SATzilla-like strategies)?
- **SATzilla-zilla:** To what extent are we able to fruitfully blend our methods and SATzilla-like strategies?
- **Feature-free portfolio SAT (minor note):** What sort of potential for performance is there in intelligently switching solvers depending on time elapsed? (Existing methods often largely commit to a single solver while sometimes making limited/heuristic use of pre-solvers and "backup" solvers.)

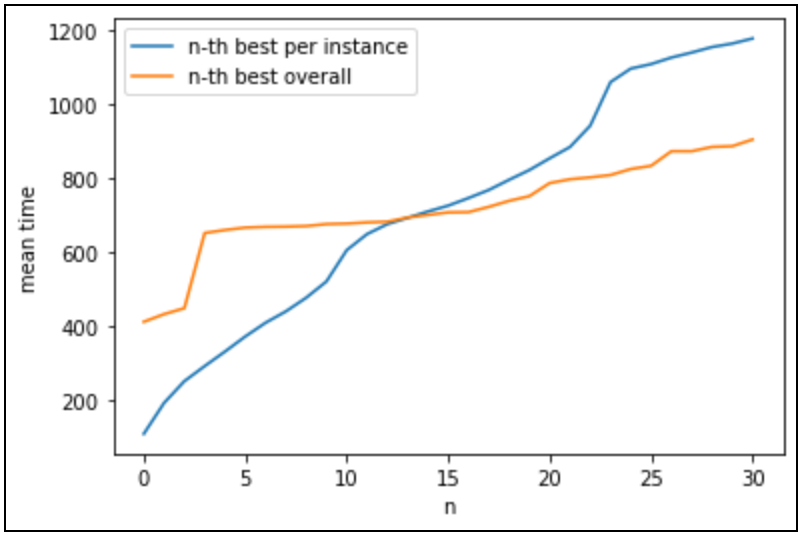



**Figure:** Comparison of $sort(E[Rts])$ (yellow) and $E(sort(Rts))$ (blue) where $Rts$ are the individual solver runtimes. Note that a portfolio method can, in general, do no better than $E(sort(Rts))[0]$ (since it has to run a solver) and should do no worse than $sort(E[Rts])[0]$ (since one could just always run the best solver instead).
comparison.svg

## Approach

### Overview

First, we compute the 1 feature group and runtime first computed by MBNG.
Given this feature group and runtime, a classifier [6] decides which RL model to continue running:

1. Monte-carlo bias-reduced n-step greedy (MBNG) [2]
2. Double deep Q network (DDQN) [3]
3. Supervised portfolio approach (SPA) [4]

All methods use SAT solver/feature data, which we preprocess [1]. MBNG depends on the data indirectly through a joint distribution [5], which we learn.

Pursuant of assessing the potential of Feature-free portfolio SAT, we implement a simple time-discretized greedy strategy [7].

### Data/Preprocessing [1/7]

We source our data from the data that was used to train the University of British Columbia team's 2012 version of SATzilla. This original data consists of four spreadsheets containing information on the performance of various solvers, feature values, and feature computation times of CNF instances. The spreadsheets are separated based on the CNF instance domain (e.g. randomly generated, industrial-like, handmade, mix of all). The separation is used to exploit domain knowledge to improve predictive power; nonetheless, our goal is to design a

general-purpose predictor, so we deal mostly with the data that includes a mix of CNF instances from all domains.

Our classifier uses the total SATzilla runtime per instance to train, but this information is not a part of the original UBC data. Thus, we simulate the SATzilla classifier, which chooses two solvers given the feature values of a CNF, using the CNF feature values that exist in the original data. Finally, we find the corresponding solver times for each instance and sum appropriately to get the total SATzilla runtime. Due to potential feature computation failures, the UBC team sets the feature runtime of these failed feature computations as -512. We disregard these values in our SATzilla runtime computation as their negative numerical value makes no sense.

## Monte-carlo bias-reduced n-step greedy [2/7]

MBNG is an approximation an optimal solution to the problem:

The learned joint distribution is
$P(F_1 = f_1, ..., F_{n_f} = f_{n_f}, R_1 = r_1, ..., R_{n_s} = r_{n_s}, T_1 = t_1, ..., T_{n_f} = t_{n_f})$
where $F_i$ are the feature group values, $T_i$ are the feature group compute times, and $R_i$ are the solver runtimes.
Given a set $I$ of $(F_i = f_i, T_i = t_i)$ pairs:
We define
The expected time it will take to run solver $i$:

$$exploit_i = E[R_i|I] = \int_{r_i} P(R_i = r_i, I)\, r_i$$

The expected time it will take to compute feature group $i$:

$$compute_i = E[T_i|I] = \int_{t_i} P(T_i = t_i, I)\, t_i$$

The expected time remaining if we decide to compute feature i is the sum of feature computation expectation and the expectation of the time remaining once we have computed the feature:

$$explore_i = compute_i + \int_{f_i}\int_{t_i} P(F_i = f_i, T_i = t_i|I)\, timeremaning(I \cup \{\} (F_i = f_i, T_i = t_i))$$

The expected time remaining until we have solved the SAT instance (including feature computation time) is the minimum of:
$timeremaining(I) = min(exploit_i \cup explore_i)$

The optimal choice we should make next is the choice that leds to the minimum remaining time:
$choice(I) = argmin_{choice}(exploit_{solver\ choice} \cup explore_{feature\ choice})$

The exact algorithm is:
$I = \{\}$
Until solver is chosen:
Make $choice(I)$. Obtain $(F_i = f_i, T_i = t_i)$. $I = I \cup (F_i = f_i, T_i = t_i)$.

MBNG approximates choice(I) by
(1) approximating the integrals in $exploit_i$, $compute_i$, and $explore_i$ with conditional sampling from the learned joint distribution. (Hence Monte-carlo.)
(2) approximating the $timeremaning(I)$ which is n recursive (feature-computing) steps after the target as simply $min(exploit_i)$. (Hence, n-step greedy.)
The algorithm admits as parameters an array of arrays: For each recursive step, the 3 desired sample sizes to use. Note that the exact algorithm

We assess the time complexity of the MBNG. We later use this for cross-validation hyperparameter tuning (so we can compare parameter settings which yield the same runtime). Let us denote the recursion (or branch) depth as $N$ and each of the 3 sample sizes (assuming constant between recursive steps) as $n_{exploit}, n_{compute}, n_{explore}$. The recursive tree branches twice with each recursive step, $n_f$ times for the features and $n_{explore}$ for sampling for each feature. Thus, each level in the recursion tree at depth $k$ has $(n_f\, n_{explore})^k$ nodes. In the overall graph, the total number of nodes is thus

$$\sum_{k=0}^{N-1} (n_f\, n_{explore})^k = (n_f\, n_{explore} - 1)^N / (n_f\, n_{explore} - 1) \;=\; O((n_f\, n_{explore})^N)$$

At each node, we sample $n_{exploit}$ times for each of the $n_s$ solvers and $n_{compute}$ times for each of the $n_f$ features: $n_{exploit}\, n_s + n_{compute}\, n_f$ in total. Thus, the overall complexity is

$$O((n_f\, n_{explore})^N (n_{exploit}\, n_s + n_{compute}\, n_f)$$

If the joint distribution is MVG, we use closed-form conditional distributions and means - improving performance substantially.

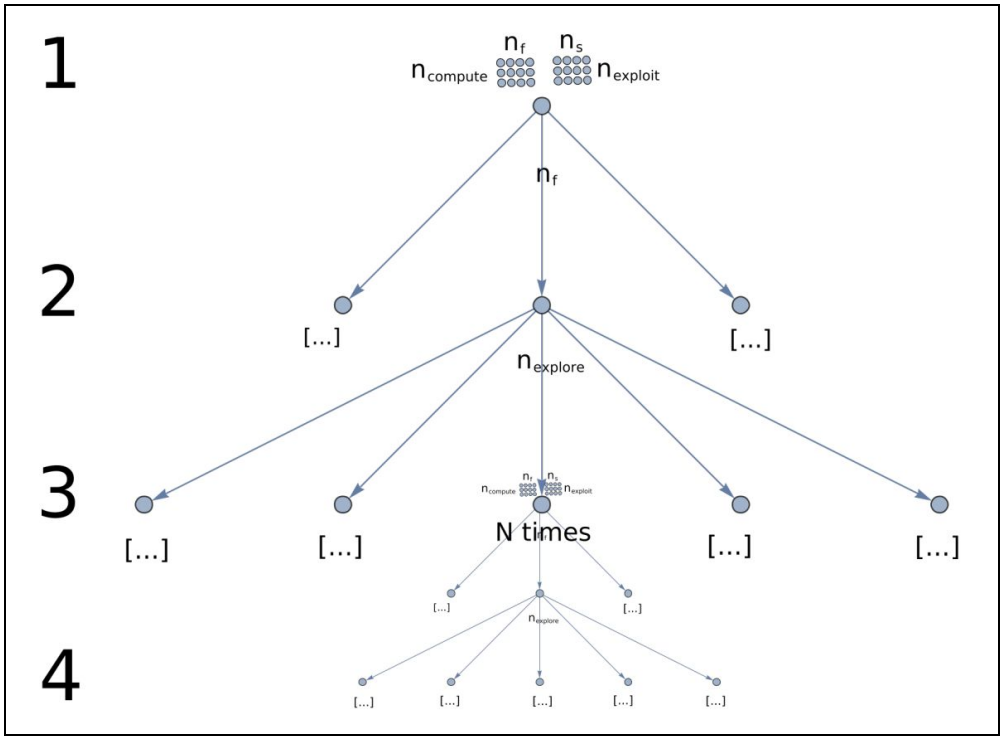



**Figure:** Visualization of MBNG process. 1. Expected feature time and solver time estimation. 2. For each feature: 3. Perform a number of samples of the feature value/time. For each: 4. Go back to step 1 (a total number of $N$ times).

Since we use $min(exploit_i)$ to approximate $min(exploit_i \cup explore_i)$ at depth $N$, the above method described produces a biased estimate of $timeremaining(I)$ for any $n's$ when $N$ is too small. In particular, $explore_i$ is systematically underestimated since $min(exploit_i) \geq timeremaining(I)$ while our estimate of $exploit_i$ is unbiased. Thus, the method should be expected to favor exploitation over exploration unduly. To account for this, as well as to introduce some meaningful model flexibility, we artificially increase the appeal of exploration options by introducing a coefficient of exploration. (Hence, bias-reduced.) This si not too different from exploration/exploitation strats (#). That is, we let
$timeremaining(I) = min(exploit_i \cup k\ explore_i)$
where $k \in (0, 1]$
We choose $k$ by cross-validation.

*Summary of justification*: MBNG implements a minimally biased approximation of an optimal algorithm.

## Double deep Q network [3/7]

Q learning uses the Bellman equation to learn the Q-value of state-action pairs by bootstrapping:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \, max_a \, Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

An action-state Q-value is the expected immediate reward if we pursue the action (from the state) plus the expected reward if we continue following the Q learning policy (from the subsequent state). Here, a Q-value represents the expected time remaining if we pursue the corresponding action.
The agent chooses the action with the highest Q-value.

In deep Q learning, a neural network (the target network) approximates the Q-value function, mapping input states to Q-values for actions. The loss function is taken from the Bellman equation and minimized with gradient descent. For improved sampling efficiency, past experiences (state,action,reward,next state) are stored and reused.

A double deep Q learning is a modification of deep Q learning which uses an additional network to prevent optimistic bias which arises from bootstrapping (since giving optimistic predictions of future state Q-values also minimizes loss). In particular, a separate prediction network is used to predict immediate rewards, the parameters of which are transferred to the target network periodically.

In our application:
The target network (2 hidden layer 30-30 dense feedforward) is inputted a vector which includes the values of all features (in all groups) as well as feature compute times. Unknown values are represented by a distinct arbitrary value.

The target network outputs a vector of estimated Q-values for possible actions. To encourage exploration, during training a random valid action is chosen with probability $\varepsilon$ (which decreases over time). In order to discourage disproportionate learning of exploitation-action Q-values (since there are more solvers than feature groups), we let the random action be equally likely an exploration as an exploitation. With probability $1 - \varepsilon$, the option with the highest q-value is chosen.
The reward at a experience is taken to be the negative time immediately incurred: the feature compute time for exploration actions and the solver runtime for exploitation actions.

During evaluation, we let $\varepsilon = 0$.
Furthermore, analogous to with MBNG, we introduce a coefficient of exploration during evaluation which artificially increases/decreases the Q-values for exploration actions; Here the decision is motivated by the potential for systematic exploration/exploitation bias due to the asymmetrical way the exploration/exploitation actions function in training (exploitation occurs less frequently, there are more exploitation options, etc.).

*Summary of justification*: DDQN generally has excellent performance in complex multi-step learning tasks with ample data (like here) while remaining quick to evaluate (critical for a solver).

## Supervised portfolio approach (SPA) [4/7]

Although we present new approaches for predicting the best solvers, these approaches are not always better than the original SATzilla 2012 algorithm. Thus, we have a problem of deciding which of our approaches to execute to get the best performance. Looking at this problem through a supervised learning approach, we precompute to determine the best algorithm for each instance in our training set and properly assign labels. Then, we train a ridge classifier using the feature values and labels for each instance in our training set. Note that even though the original UBC dataset does not contain the SATzilla total runtime for each instance, we simulate SATzilla during preprocessing and use the solver runtimes provided in the UBC dataset to determine the desired value. As discussed later in this paper, the trained ridge classifier produces decent results that give us a non-negligible advantage over running the original SATzilla algorithm.

## Learning the joint distribution [5/7]

In each decision-making step of MBNG, some subset of the full feature set has already been computed for the given SAT instance. In order to estimate the expected time remaining, we need the ability to condition our predictions of feature computation times and solver runtimes on arbitrary subsets of computed feature values. To do this, we approximate the joint distribution of feature group values, feature group computation times, and solver runtimes using a parametric model, and then sample from the conditional distribution of unknown variables at each step. Since the number of features may be large, it is computationally intractable to estimate the marginal densities through numerical integration. Thus, we model the joint distribution as Gaussian mixtures so that the parameters of each conditional distribution can be calculated through a closed-form expression. We learn the parameters of the joint distribution from the SAT data by maximum-likelihood estimation.

The parameters for the Gaussian Mixture model were learned using sklearn's `GaussianMixture` module, which uses the EM algorithm to learn the mean vector and covariance matrix for each component of the mixture as well as the weight of this component, given a number of components, some specifications for the form of the covariance matrix and the data. We tuned the parameters of the mixture model using 5-fold cross-validation. The parameters we tuned as well as the range of values considered for each parameter are as follows:

- `n_components`

  An integer specifying the number of components in the mixture model. We considered all the integers from 20 to 30, as during preliminary tuning it was found that preliminary this parameter always fell somewhere between 20 and 30.
- `covariance_type`

  This parameter is categorical and determines what restrictions are placed on the covariance matrices. Since it was computationally feasible, we considered all possible values for this parameter. If 'full', each component is allowed its own general covariance

matrix. If ‘tied’, all components share the same general covariance matrix. If ‘diag’ each component has its own diagonal covariance matrix and if ‘spherical’ each has its own single variance (diagonal covariance matrix with entries on the principal diagonal being identical).

- `reg_covar`

  Non-negative regularization added to the diagonal of covariance. The purpose is to ensure numerical stability - we decided to try different values to see if this parameter could somehow prevent overfitting as well. We considered the range `np.logspace(-10, 1, 20)`.

### Classifier [6/7]

At a high level, the classifier is meant to determine which of our three approaches is most worth executing— MBNG, DDQN, or the original SATzilla method (of which we developed a heuristic to approximate). Given a single pre-computed feature group (ideally the first group which will always be computed), it seeks to predict which of the three approaches runs the fastest, at which point it returns its selection. This is then executed, and we analyze performance compared to the pre-existing SATzilla model.

### Feature-free portfolio SAT [7/7]

The basic approach is, at every timestep, to greedily continue running the solver which has the greatest probability of yielding a solution in the next marginal time step. We justify the approximate optimality of such an approach due to the uncorrelatedness of solver runtimes and consistent diminishing returns over time across solvers; the probability of yielding a solution should be largely independent between solvers and not improve at a later time. To minimize bias, we make empirical estimates rather than using the fitted joint distribution. To minimize the variance in such estimates, we introduce a parameter $p$ which controls the percent of earliest future instances to average. In particular, having run the solvers for times $ts$, we choose the next solver to compute for a marginal time step $dt$ by choosing the solver with maximum:

$E(min_p(Rt) - t | Rts > ts)$

where $Rt$ in the solver runtime, Rts is all the solver runtimes, t is the amount of time run the solver thus far, and $min_p$ is $1$ if $Rt = rt$ is in the lowest $p$ percent of all $Rt = rt$ (conditional on $Rts > ts$).

## Experiments and results

### Joint distribution [1/5]

We measured goodness of fit by the lower bound on the log-likelihood yielded by the EM algorithm. Since runtimes are truncated at 0, we wanted to fit to the (natural) log of the data, as these values were more likely to be distributed normally. But due to the fact that some features occasionally had negative values, we tried applied min-max scaling to the range (0.1, 1). A figure visualizing the results of our gridSearch is shown below (the best model had a score of

109.04).However, the resulting mixture model (`n_components`=26, `covariance_type` = ‘diag’, `reg_covar` = 1e-10) seemed to perform worse than a simple stand-alone multivariate gaussian in practice.

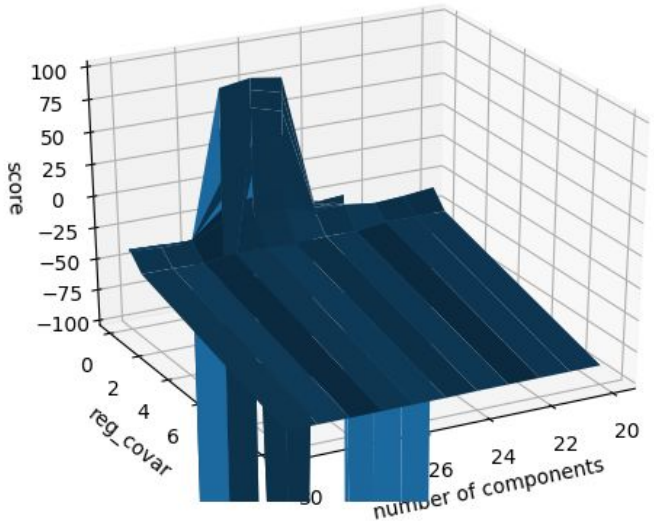



**Figure:** Visualizing the grid search by plotting the score of the model versus the number of components of the distribution and the covariance matrix regularization strength. We see that performance increased with decreased regularization strength and that the sweet spot for number of components was the range 24-28.

Because of the MinMax scaling, it was possible for this model to sample negative runtimes and we suspected that this might have been the reason for bad performance. However, various transformations we attempted did not lead to higher scores for the fitting of the distribution or to better algorithm performance.

## Tuning and visualizing DDQN [2/5]

We use 1-fold (90% train, 10% test) cross-validation to select the test performance optimizing coefficient of exploration.

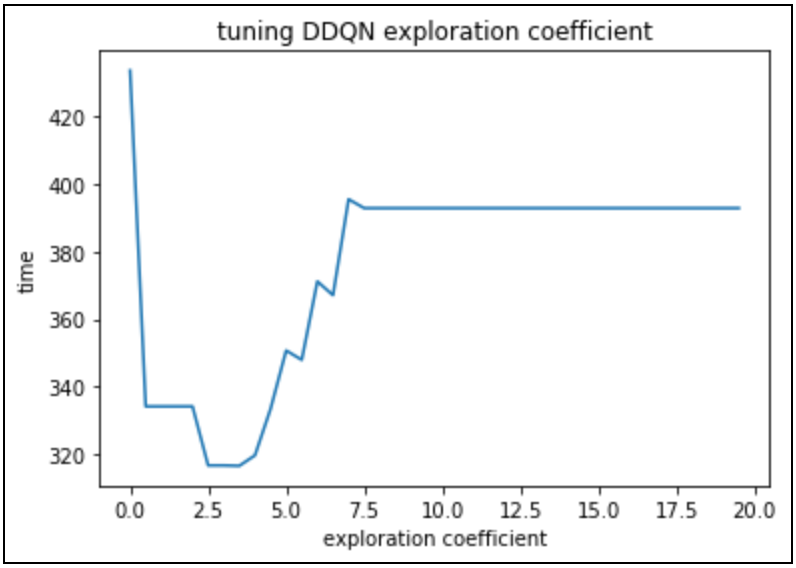



**Figure:** We find $exploration\ coefficient^* = 7.5$ (favoring exploration over the base model). Note that the right plateau approaches around the overall best solver average runtime.

We also spend some time consecutively tuning iterations and min exploration actions. We obtain $iterations^* = 10^4$, $min\ exploration\ actions^* = 0$ (i.e. no use).
We visualize the decisions made by the DDQN.

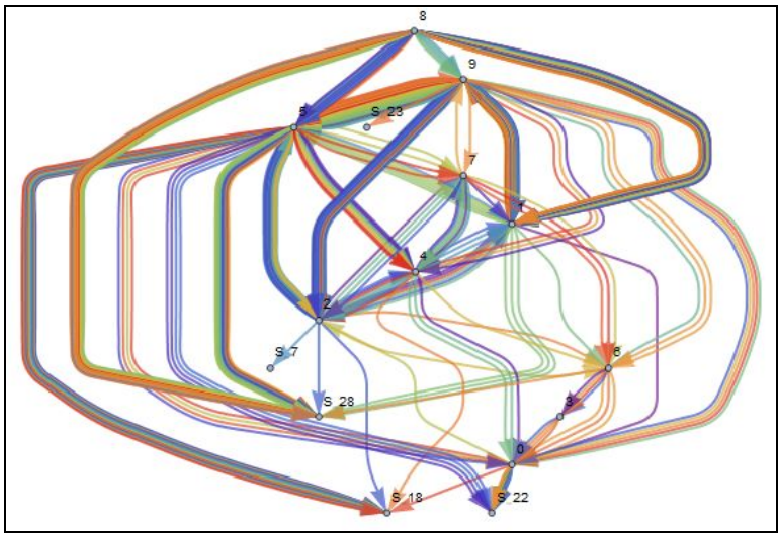



**Figure:** Layered digraph embedding of 100 instance decision paths for DDQN. Features are represented by nodes labeled {number}. Solvers are S_{number}. Instances are directed paths of arbitrary colors; they pass through features they compute and end at solvers they run.

We summarize the tuned DDQN parameters:

| iterations | exploration coefficient |
|---|---|
| $10^4$ | 7.5 |

Due to the random nature of DDQN training (an substantial empirical variation in performance), 10 runs were performed and the run with the best out-of-sample performance was chosen.

## Tuning and visualizing MBNG [3/5]

We choose which joint distribution form (fitted in the previous section) to use for conditional sampling.

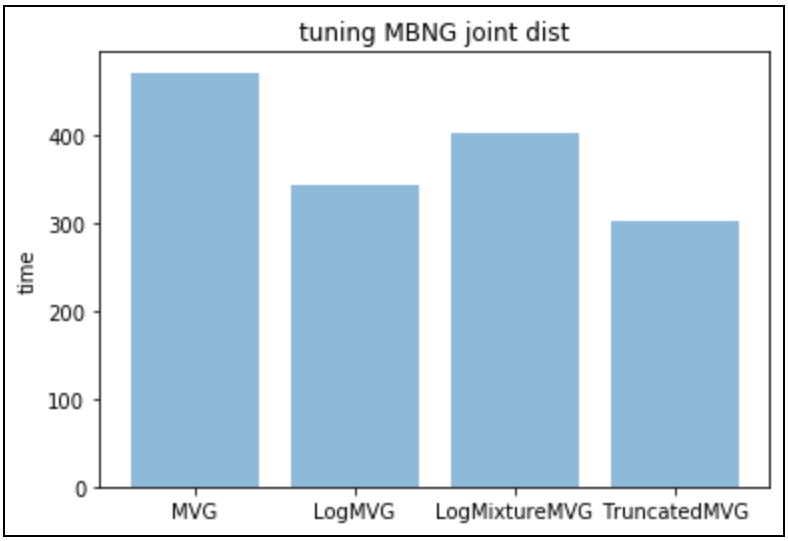



**Figure:** Comparison of MBNG mean overall runtime for various distributions. Results are produced using tuned monte-carlo parameters. We choose truncated MVG.

Unlike the DDQN, MBNG has parameters we are interested in which substantially affect runtime while constantly improving performance (in an asymptotic way): the monte-carlo parameters. We tune them while keeping runtime roughly fixed.

First, we tune the time-linear $n_{exploit}$, $n_{compute}$. We find the performance peaks at around $n_{exploit}$, $n_{compute} = 20$ and in general that the parameters are not super-critical performance-wise .
We are more concerned with the exponential and polynomial parameters $N$, $n_{explore,1}$, $n_{explore,2}$.
Using the derived MBNG time complexity expression, we devise near time-equal strategies to test various tradeoffs between $N$, $n_{explore,1}$, $n_{explore,2}$

| setup | $N$ | $n_{explore,1}$ | $n_{explore,2}$ | time |
|---|---|---|---|---|
| 1 | 1 | 250 | N/A | 323.32 |
| 2 | 2 | 5 | 5 | 363.29 |
| 3 | 2 | 8 | 3 | 482.55 |

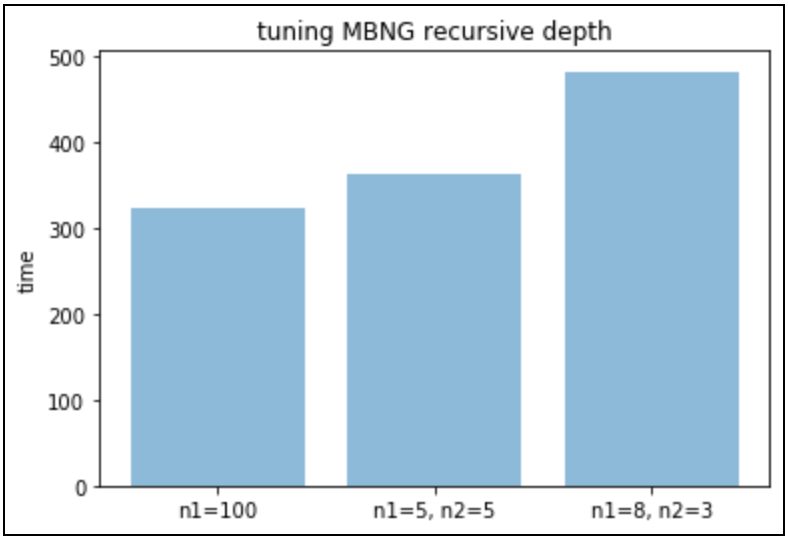



**Figure**: We choose setup 1. We are surprised that setup 2 outperforms setup 3; we reasoned that accuracy in more distant considerations (i.e. $n2$ related) would be less valuable. We hypothesize that the result is due to the higher variance for $n2$ samples contributing to more inaccurate estimates. LogMVG is used for this test.

We summarize the tuned MBNG parameters:

| $N$ | $n_{explore}$ | $n_{exploit}$ | $n_{compute}$ | exploration coefficient |
|---|---|---|---|---|
| 1 | 250 | 20 | 20 | 1 (i.e. no use) |

## Classifier [4/5]

The first task here is to determine which single feature group of the ten potential ones is the best at predicting the runtime for each of our three approaches. Ultimately, ridge performed the best and the results are below:

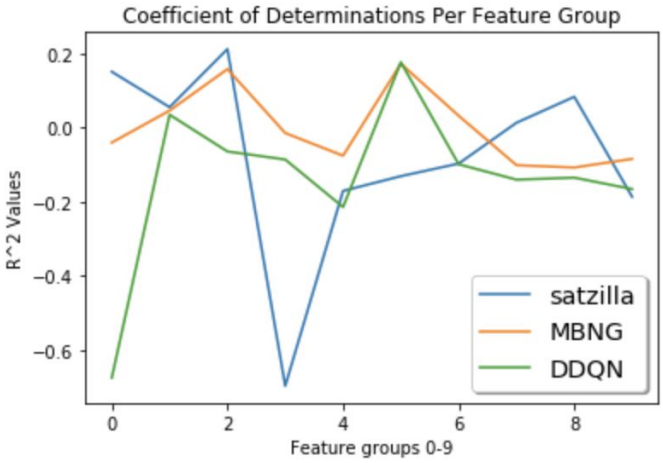


*Figure indicating best coefficient of determination values for using a single feature group to predict runtime for each of our three approaches.*

Feature group #2 (indexing by 0) is the best for predicting satzilla (SPA) runtimes, while feature group #5 is better for predicting MBNG and DDQN runtimes. Regardless, the coefficients of determination (resulting from the "score" function from the Ridge Regression library) are fairly low— hovering around 0.2. As a result, these predictive models are not optimal— however if we were to only compute a single feature group, computing groups 2 and 5 (by 0 index) will be significantly more optimal than other alternatives. The performances are as follows when predicting

```
Our algo vs. SATZ accuracy: 0.52
DDQN vs. SATZ accuracy: 0.62
Our algo vs. DDQN accuracy: 0.69
```

*Figure elucidating the performances when predicting against each other.*

When treating this as a binary issue and only comparing two approaches at a time against each other to find which one is predicted to be faster, the performances, despite the low coefficients of determination, are actually decent. To ensure using our 3 approaches is the best, we compute the average actual runtime of our model's selection for the algorithm. The model selects the approach it deems fastest (call A: SATzilla-esque, B: MBNG, C: DDQN) and then returns the actual runtime of its algorithm selection. We add weights of 1, 7, and 12 to each of our runtimes such that in order to choose the SATzilla-esque option, $1*SAT_{runtime} < 7*MBNG_{runtime}$ and $1*SAT_{runtime} < 12*DDQN_{runtime}$, etc. The results for average runtimes in seconds are below:

```
Avg. Runtime ABC: 283.93138149076566
Avg. Runtime AB:  287.1193521363812
Avg. Runtime BC:  325.5012467650398
Avg. Runtime AC:  287.46596055054977
```

Avg. runtime ABC represents the average runtime per instance when we predict on all three and follow the threshold rules. AB represents when we only predict on A and B (and thus the algorithm chosen can only be A,B). The results indicate that ABC outperforms any 2-subset of the three, indicating that choosing from all three algorithms yields a better result than simply sticking with two.

## Tuning and visualizing feature-free portfolio SAT [5/5]

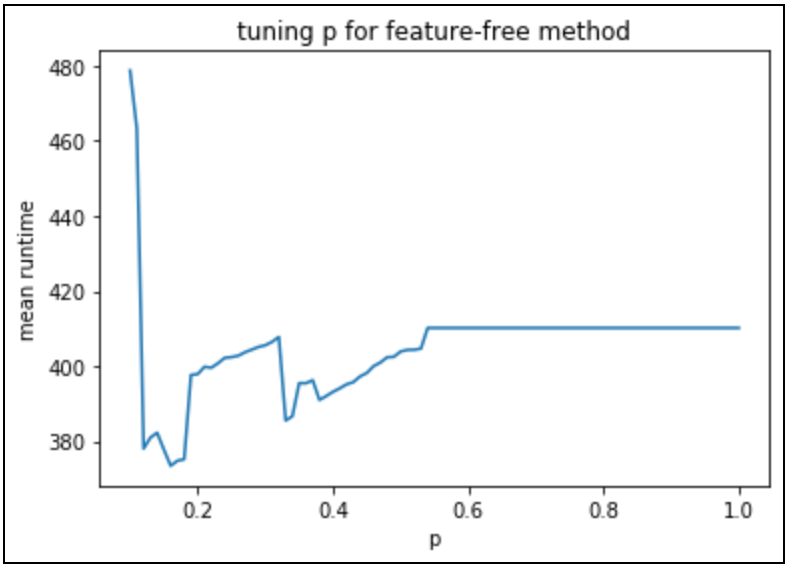



**Figure:** We tune $p$ by cross validation, the proportion of minimum runtimes to average when estimating how desirable a solver is to continue running. We choose $p = 0.16$. Note that, for $p = 1$, the method is equivalent to choosing the best overall solver (mphaseSATm). On this dataset, we find that the method only runs mphaseSATm for any $p \geq 0.54$ as well.

We find that the timestep $dt$ generally only improves performance (with diminishing returns); we choose $dt = 0.01s$ to minimize training time.

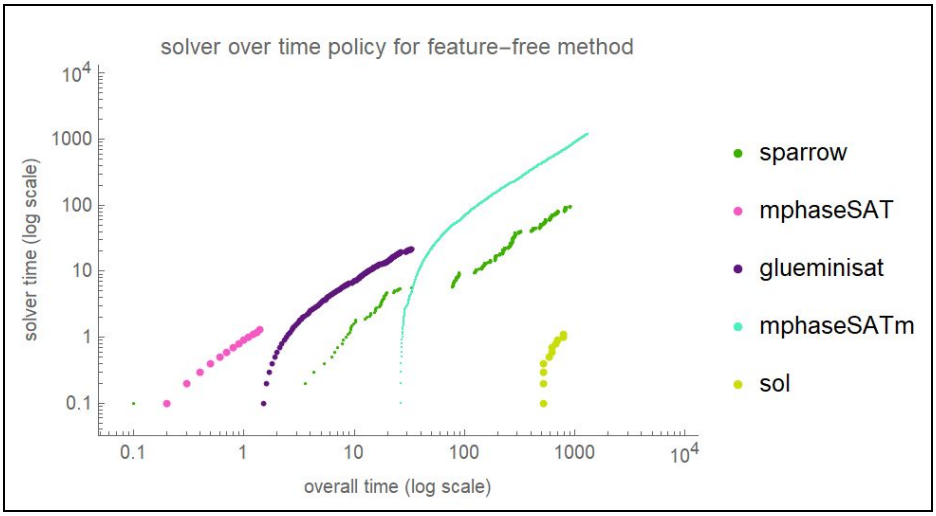



**Figure:** We visualize the feature-free method's policy, which is solely a function over overall time. Notice that very often solvers are alternated between rapidly.

## Discussion and conclusion

**Better joint distribution, worse performance [1/4]**

A significant difference was observed between how well the joint distribution fit the data (as measured by $P(out\ of\ sample\ data|distribution)$ - “fit”) and how well MBNG with that joint distribution performed on the SAT instances (as measured by mean overall runtime - “runtime”). We identify two main features to account for this: (1) the log transformation and (2) positive density for negative feature and solver times, both of which were associated with increased fit but increased runtime. We suspect that feature (1) caused distortion in expectation of sampled runtimes and (2) caused the posterior distribution to be inaccurate due to the absence of negative runtimes in training.

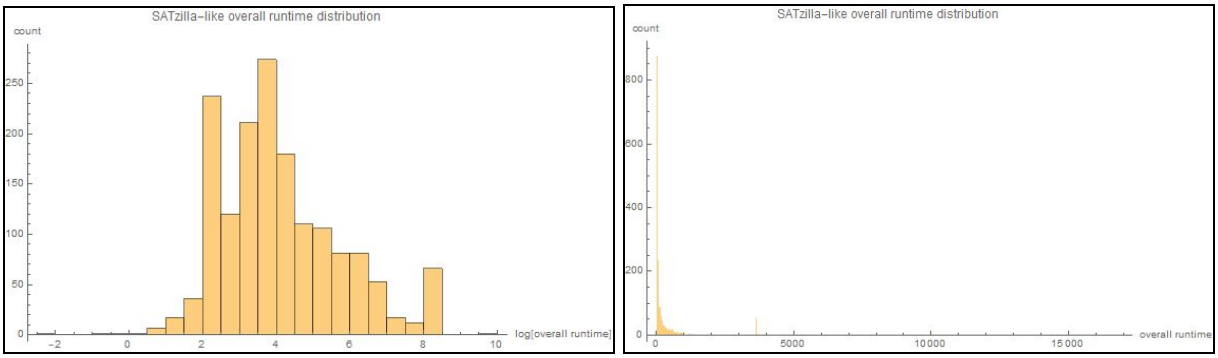
**Tradeoffs between MBNG, DDQN, and SATzilla-like (SPA) [2/4]**

Training and evaluation

For close to peak overall performance: MBNG is quicker to train (~2000x), slower to evaluate (~1000x), gives better performance than DDQN (~5%, mean 303.21s vs 319.28s per instance). Both performances are roughly equivalent to running the 4-5th best solver per instance (of 31 solvers). Due to the importance of valuation time to the application, we speculate that model-free methods like DDQN may have more potential. However, DDQN showed less performance increase in increasing training time than MBNG did in increasing evaluation time - so alterations to the method or more exhaustive tuning (their are numerous DDQN parameters, including the net architecture) may be appropriate.

Skew

It is well established empirically that a minority of worst-case instances in NP-hard problems can be disproportionately detrimental to performance. We observe this phenomenon in portfolio SAT, with overall runtimes being highly right-skewed (approximately log-normal).




**Figures:** Overall runtimes for SATzilla-like are approximately log-normal. Similar is observed for other methods.

We analyze the minority detrimental cases by distinguishing between runs where a timeout occurs in a solver. Among the the methods, a tradeoff is observed between the proportion where no timeout occurs and the runtime when no timeout occurs. We propose that this tradeoff could be exploited in cases where there are more or less worst-cases - or where solving every instance exactly is not critical.

| Method | Mean total runtime (s) | Proportion where no timeout occurs | Runtime when no timeout occurs (s) |
|---|---|---|---|
| DDQN | 303.21 | 0.81 | 103.93 |
| MBNG | 319.28 | 0.84 | 126.49 |
| SATzilla-like | 304.41* | 0.94 | 122.60 |
| mphaseSATm | 410.24 | 0.75 | 150.56 |

**Table:** Performance statistics for various SAT portfolio methods. _, the best overall solver is included for comparison. *For solvers which timed-out (runtime of 1201s), we treated runtime as 1201s. SATzilla-like treats the value as $\infty$, so runs another solver. Thus, 304.41s is an upper-bound. A very loose lower bound is given by capping SATzilla-like total runtime to 1201s, yielding 183.40s.

**Feature-free potential [3/4]**

The method achieved an average runtime of 373.45s, as expected significantly worse than the feature-based portfolio methods. However, this result is still 9.85% better the best solver. As such, we propose that feature-free method could be used as a backup solver (when feature computation errors) and pre-solver (before features are computed). Furthermore, we identify potential in a feature-free, feature-dependent hybridig where solvers may be halted and continued dependent on computed features. Finally, the rapid oscillation between solvers exhibited by our method suggest the suboptimality of the sequential pre-solver approaches seen in SATzilla. We acknowledge a limitation of our work in that we did not account for any potential overhead in interrupting and recontining solvers, although we feel confident this will not eliminate the value of the approach (even very coarse time steps showed value) and suspect that such overhead may be reducible (e.g. through running solvers on emulators).

**Classifier conclusion [4/4]**

The ridge regression models predict the raw runtimes with a low coefficient of determination, but when it comes to performance of our approach— selecting 1 of 3 algorithms— this actually outperforms SATzilla 57% of the time. This approach of instituting 3 meta-algorithms and selecting the best of the 3 to carry out seems to achieve a decent amount of success. More importantly, the classifier's method of selecting which algorithm to choose (factoring in weights) yields a better composite result, than sticking to any subset of 2 of the 3 methods.

**References**

[1] 2019 https://arxiv.org/pdf/1909.11830.pdf
[2] 2008 http://homepages.laas.fr/ehebrard/papers/aics2008.pdf
[3] 2011 https://www.researchgate.net/publication/221633386_Algorithm_Selection_and_Scheduling
[4] 2019 https://arxiv.org/pdf/1903.04671.pdf
[5] 2019 https://rlgm.github.io/papers/32.pdf

[6] 2017 https://arxiv.org/pdf/1709.07615v2.pdf
[7] 2009 http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/SATzilla2009.pdf