Optimizing Data Pre-processing Transformations with Reinforcement Learning

by

Brandon D. Finley

B.S., University of Colorado, Boulder, 2021

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Applied Mathematics

2022

Committee Members:

François G. Meyer, Chair

Maziar Raissi

Claire Monteleoni

Finley, Brandon D. (M.S., Applied Mathematics)

Optimizing Data Pre-processing Transformations with Reinforcement Learning

Thesis directed by Prof. François G. Meyer

In this work, we use Reinforcement Learning (RL) to optimize the data pre-processing transformations in a machine learning pipeline given a dataset $\mathcal{X}$, child algorithm $f(\cdot)$, and action space $\mathcal{A}$. Inspired by Effective data pre-processing for AutoML [4] and Learn2Clean [1], we construct a model that 1) does not specify a data pre-processing pipeline structure in advance and 2) does not depend on transformation specific rules or empirical calculations across multiple datasets. Using a simple policy optimization scheme, we produce comparable results to [4] across multiple datasets from the OpenML-CC18 benchmark suite and with Naive Bayes (NB) as our child algorithm. This was accomplished by only finding an optimal order of actions sampled from the action space $\mathcal{A}$, where the parameters of each action were kept to their default values. We hope that this model can serve as a basis for future projects, including the study of how such data transformations affect the manifold structure as well as implementing a conditional aspect to our model to make it more efficient.

Dedication

To my family, thank you for your unending support.

Contents

Chapter

Tables

Table

Figures

Figure

Nomenclature

$\mathcal{A}$	Action space
$\mathcal{X}$	Tabular dataset
π_θ^*	Learned policy
τ^*	Learned trajectory
τ_0	Baseline trajectory
θ	Controller's weights
a_t	Specific action chosen at time t (e.g. StandardScaler)
a_t^i	Specific action that corresponds to the i^{th} action in $\mathcal{A}$ at time step t (e.g. a_t^3 = "Min Max Scaler")
a_{tk}	Specific action chosen at time t for trajectory k (e.g. StandardScaler)
$f(\cdot)$	Child algorithm
M	Number of episodes
N	Number of actions in action space $\mathcal{A}$
T	Number of time steps

0.1 Introduction

In a machine learning pipeline, we generally have two main components: the data and model. The data section contains all the downloading of the data, cleaning, pre-processing, and feature engineering. The model section contains model selection, training, and tuning. It is well known that both of these sections are dependent on each other. For this reason, constructing the pipeline is considered to be a mainly human-centered component of design, as is it very data and model dependent and has multiple solutions. Due to this, the idea that one can use machine learning itself to help automate a machine learning pipeline came out of existence and was coined "Automatic Machine Learning" (AutoML) [5].

Among the most frequently used ways to implement this is to use Reinforcement Learning (RL) [7]. RL can be defined as *learning what to do—how to map situations to actions—so as to maximize a numerical reward signal* [9]. In this setting, we can train a model to find a machine learning pipeline by using a performance metric as the reward signal. The model component of the machine learning pipeline that provides the reward signal is called the *child algorithm*. This process is done by using a controller network to sample a certain machine learning pipeline and then test it to get the reward signal. Now, while we consider the entire pipeline to be important, most work focused on the model section - namely how to build the best architecture. However, recent efforts have put more focus on the data section of the pipeline as we have come to understand that each section is extremely important (especially because data pre-processing seems to be the most time consuming aspect of a data scientist's job).

In this work, we will construct a scalable model that automatically finds a pre-processing sequence for a dataset $\mathcal{X}$, child algorithm $f(\cdot)$, and action space $\mathcal{A}$. More specifically, we use RL to construct a machine learning pipeline entirely made from pre-processing transformations that maximizes the accuracy of a child algorithm. We create a more general model as it does not require specifying a pipeline in advance or empirical results across multiple datasets. We were able to get comparable results to an existing paper [4] with the contribution that our model is more general

and scalable. We aim to augment this further for better accuracy and efficiency and also to use it as a tool for research purposes.

0.2 Related Work

In this section, we will briefly go over the most influential papers for this work. An illustration of the relevant papers on a timeline can be seen in Figure 1. Within each subsection, we go over the background of each paper as well as the contribution or difference to our approach.

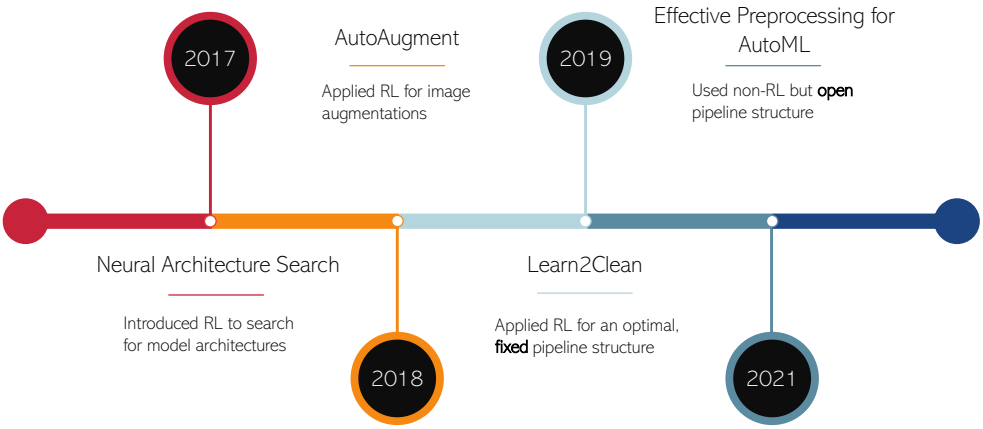

Figure 1: A Timeline of Related Work

0.2.1 Neural Architecture Search

In Neural Architecture Search (NAS), the aim is to learn an architecture that maximizes the validation accuracy of a child algorithm. Instead of brute-force searching a possibly infinite combination of architectures and parameters, one can use Reinforcement Learning (RL) to expedite the process. Specifically, one can use a controller [12], where the architecture parameters are

encoded in the controller weights, θ. In this setting, a single forward pass of the controller samples multiple architectures for the child network. After sampling these architectures, the child network then uses them to train the child algorithm and compute the validation accuracy of the resulting model. In [12], their RL method rivaled the best human-generated architecture and was faster than the previous state of the art network. Overall, this paper's method serves as the foundation for this thesis.

0.2.2 AutoAugment

Similar to Neural Architecture Search, AutoAugment [3] uses a controller and RL for its search strategy. However, instead of learning an architecture, the goal is to learn image augmentations for a target dataset $\mathcal{X}$ of images, such that the the child algorithm $f(\cdot)$ achieves the highest validation accuracy. In their setup, the learned policy π_θ is made up of 5 sub-policies. For each mini-batch, one sub-policy is chosen at random. Moreover, within each sub-policy, there exist two transformations with a corresponding magnitude and probability. Thus, the sub-policy has 2 operations, 2 magnitudes, and 2 probabilities (6 parameters to predict) while the whole policy has $6 \times 5 = 30$ parameters to predict. In the end, they attain competitive results and show that the learned policies of one dataset can be transferred to other datasets to obtain significant improvements. This paper served as the inspiration for the thesis as it showed that RL can be used to learn transformations for the dataset also rather than just the child algorithm architecture.

0.2.3 Learn2Clean

In Learn2Clean [1], RL is used to find the best pre-processing sequence given a dataset $\mathcal{X}$, child algorithm ϕ, and quality performance metric q. In general, this paper is the closest to our approach when compared to the other methods. However, there are a few main differences: 1) the model is specific to cleaning web data, a different application, and 2) the pre-processing pipeline structure is *fixed*, e.g. for a given task, they might require that either normalization or feature engineering comes first. This then requires pre-existing knowledge about the datasets (that it is

numerical and does not require encoding). In contrast, our pipeline structure is *open*, in that no pipeline structure is specified in advance.

0.2.4 Effective Pre-Processing for AutoML

While the other related works involve RL, Effective Pre-Processing for AutoML [4] attempts to find an optimal pre-processing sequence by reducing the possible combinations through heuristics and empirical calculations. For instance, using a 3 step process, they use 1) framework-related rules, 2) heuristic rules, and 3) learnt rules to find effective pre-processing pipelines. The paper offers some examples for each type of rule: 1) encoding is required for datasets of mixed-type, and so, encoding will be required to come before other transformations; 2) normalization ought to be applied before rebalancing as it affects the magnitude of the values; 3) with the remaining possible orderings filtered from 1) and 2), an exhaustive computation is performed to determine if a particular ordering yields superior results. Our approach differs from this paper as we 1) do not depend on transformation specific rules and 2) empirical calculations. Our model does not specify a pipeline structure in advance, giving our model an *open* pipeline structure. In this work, we will be using this paper as a baseline metric to compare our results.

0.3 Method

0.3.1 Design and Algorithm

Our goal is to learn the policy, π_θ, a stochastic mapping:

$$(\mathcal{X}, f(\cdot), \mathcal{A}) \mapsto \pi_\theta$$

such that when we sample a trajectory τ^*: $\tau^* \sim \pi_\theta \implies \tau^* = \{a_1, a_2, \cdots, a_T\}$ the child algorithm's performance metric is **maximized**.

We note that $\mathcal{X}$ is a tabular dataset, $f(\cdot)$ is the child algorithm, T is the number of time steps, and $\mathcal{A}$ is the action space.

To aid in the understanding of the model, we can look at the overall design of it in Figure 2., a flowchart representing four main components of the model. This will serve as an outline to the rest of the paper providing the necessary intuition. We cover the four points with references to the rest of the paper:

(1) For a given state of our controller with parameters θ, we sample from the policy π_θ a total of M times. This produces M trajectories τ_k.

(2) Using the M trajectories, one can apply the specified transformations to the training set and then obtain the reward, R_k, balanced accuracy (Equation 12).

(3) Using all of the trajectories, we will compute a single scalar b that represents the exponential moving average of the previous trajectories (Equation 9).

(4) Lastly, we can now compute the total loss. The total loss is the arithmetic mean of each trajectory's loss, $J(\theta)$. The arithmetic mean serves as an approximation to the expectation value as its constant can be absorbed into the learning rate.

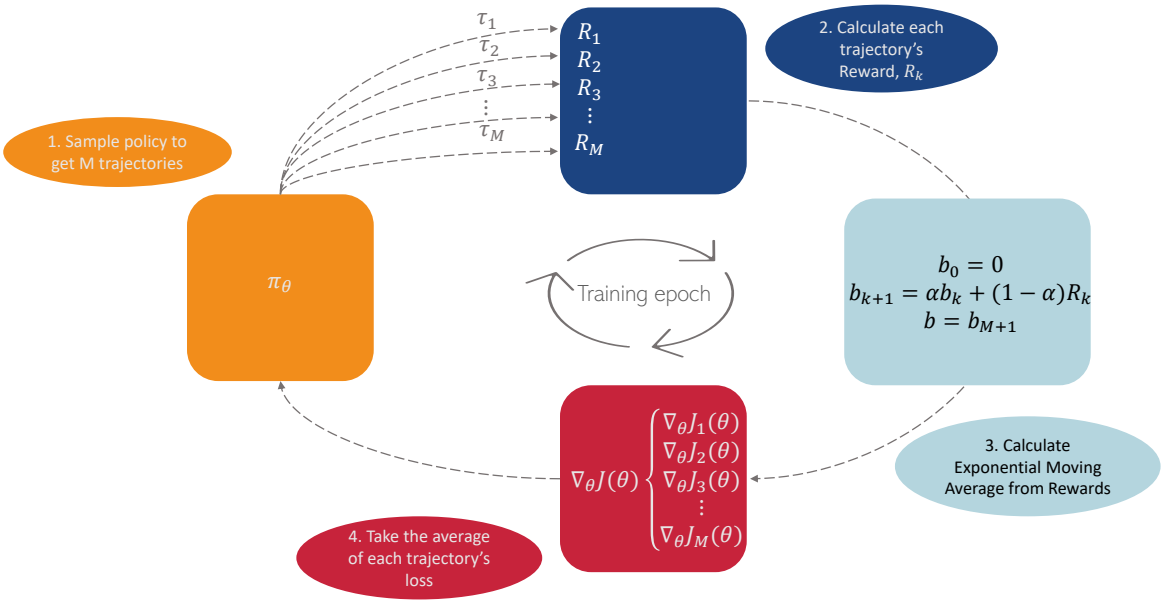

Figure 2: Overall Process of the Policy Optimization Procedure

Similar to Figure 2, we can further clarify the process by looking at some pseudocode for the overall process. This is represented in Algorithm 1.

0.3.2 Controller

In order to find the best pre-processing sequence, we need to have the ability to encode sequences into our controller. Likewise, it must be able to store sequential information from previous transformations in the sequence. Thus, we turn towards Recurrent Neural Networks (RNN) and Long-Short Term Memory Networks (LSTM) and use the same procedure as [12]. While the LSTM is considered a more widely applicable variant of the RNN, the vanishing gradient issue in long RNN's does not pose a problem due to our short sequence length.

A single RNN's cell operates by taking in a hidden state h_t and input vector x_t. The next hidden state is updated by first multiplying h_t and x_t with separate weight matrices $W_{h_t h_t}$, $W_{h_t x_t}$, then summing the values together, and finally using a non-linearlity such as ReLU or hyperbolic

Algorithm 1: Policy Optimization Algorithm

Input : X, y, and $f(\cdot)$
Output: π_θ^* (trained policy)
for each epoch do
  for each episode do
    $\tau_k \sim \pi_\theta$
    $\hat{X}$, $\hat{y}$ = apply_transformations(X, y, τ_k)
    R_k = get_reward($\hat{X}$, $\hat{y}$, $f(\cdot)$)
  end
  for R_k **in rewards do**
    $b_{k+1} = \alpha b_k + (1-\alpha)R_k$
  end
  $b = b_{M+1}$
  for R_k **in rewards do**
    J_k = trajectory_loss(R_k, b, τ_k)
  end
  $J(\theta) = \frac{1}{M}\sum_{k=1}^{M} J_k$
  $\theta = \theta + \gamma_w \nabla J(\theta)$
end

tangent. The calculation can be found in Equation 1.

$$h_{t+1} = \text{ReLU}(W_{h_t h_t} h_t + W_{h_t x_t} x_t + b_{h_t}) \tag{1}$$

In Equation 1., we have three trainable parameters: $W_{h_t h_t}$, $W_{h_t x}$, and b_{h_t}. Now, we want to use this cell so that it outputs a discrete probability distribution. This will allow us to sample a single action from this specific cell and then feed it into the next cell as the input, x_{t+1}. We can achieve this discrete probability distribution by first using a linear matrix to change the dimension of the hidden state into the same number of actions in our action space and then use a softmax function to get our probability distribution. This is the same way [12] encoded an architecture of a neural network. Formally, we use the following

$$y_t = \text{softmax}\left(W_{y_t h_t} h_t + b_{y_t}\right) \tag{2}$$

Finally, we can sample from this discrete probability distribution, y_t, to get our action a_t. We will then map this action into an embedding space: $g : a_t \mapsto x_{t+1}$, where $a_t \in \mathbb{N} \cup \{0\}, x_{t+1} \in \mathbb{R}^{Y \times 1}$. The structure of a single time step is seen in Figure 3.

Action Space $\mathcal{A}$	
Category	Action
None	Identity Map
Normalization (N)	Standard Scaler Power Transformer Min Max Scaler Robust Scaler
Feature Engineering (F)	PCA Select K Best PCA + Select K Best
Discritization (D)	Binarizer K Bins Discretizer
Imputation (I)	Simple Imputer
Encoding (E)	One Hot Encoding Ordinal Encoding
Rebalancing (R)	SMOTE

Table 1: Our Specific Action Space $\mathcal{A}$

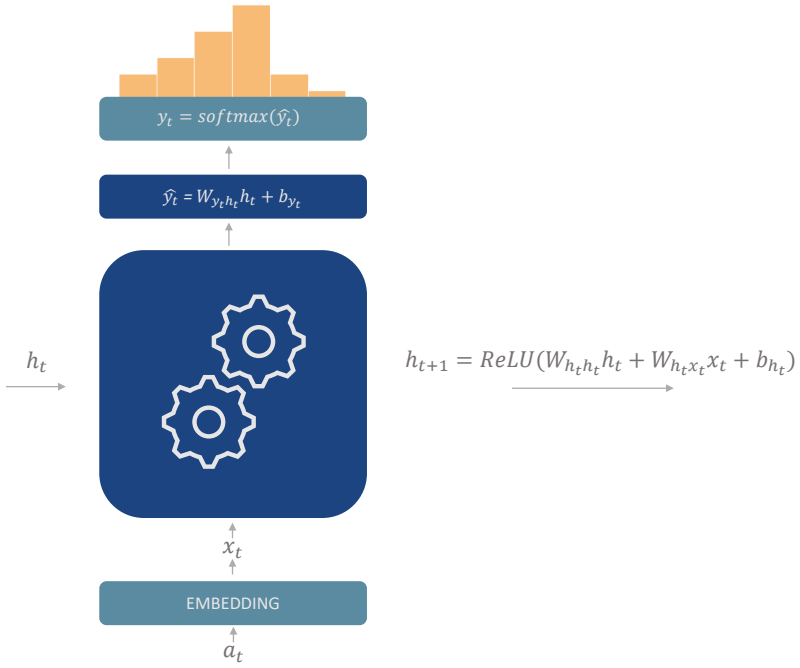

Figure 3: A Single RNN Cell

We will repeat this process T times to get our trajectory τ. Overall, all the RNN cells together form the controller. Altogether, this allows us to encode a sequence of possible transformations

inside a neural network that we can then tune with Policy-Gradient Optimization. The entire structure is depicted in Figure 4.

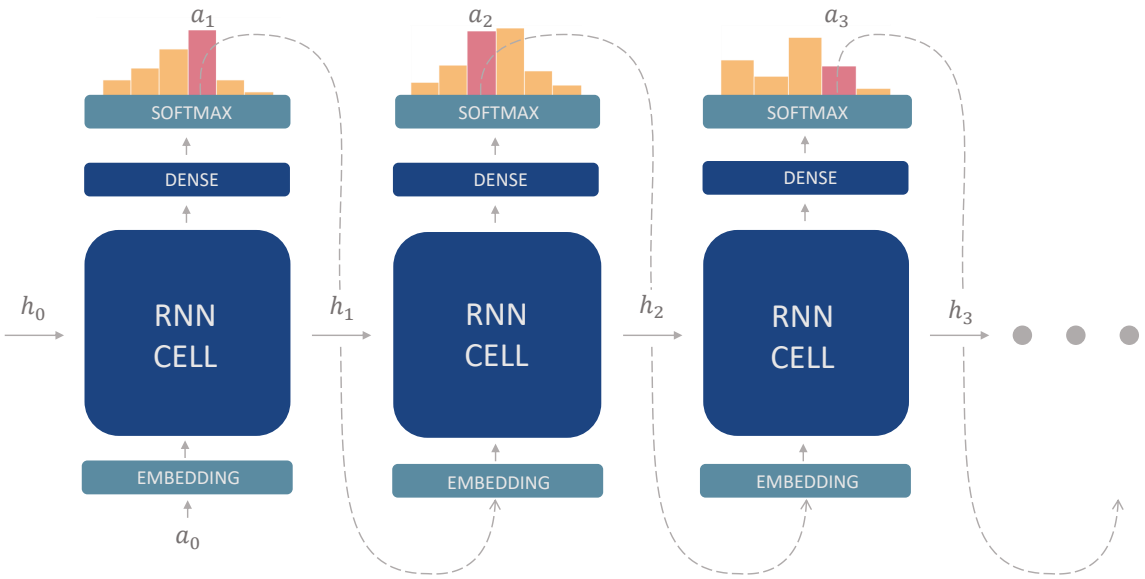

Figure 4: The Structure of the Controller

0.3.3 Policy-Gradient Optimization

To train the controller, we will implement policy-gradient methods [10]. Using the controller's weights, θ, we can equate a forward pass of the network to be the policy, π_θ. Our goal is to find such policy that maximizes the child algorithm's performance. Therefore, for a policy π_θ, we aim to maximize the expected reward over trajectories sampled. This is seen in Equation 3.

$$J(\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta}(R(\tau)) \tag{3}$$

Due to the task being to maximize the objective function 3, we effectively perform gradient ascent. Additionally, in our setup we will be using a non-constant learning that will be a function of the

epoch, w. Thus, our gradient update becomes Equation 4.

$$\theta \leftarrow \theta + \gamma_w \nabla_\theta J(\theta) \tag{4}$$

Before we continue, let us quickly review what a single trajectory looks like. Formally, the trajectory τ is the sequence that contains the state, reward, and action information at every time step. For example, a trajectory looks like the following:

$$\tau = \{s_1, a_1, R_2, s_2, a_2, R_3, \cdots s_T, a_T, R_{T+1}\}$$

Although this is the most general form, our setup is slightly different. Firstly, we only give rewards at the end of the trajectory and so we have a single reward for a single trajectory. Another way to look at this is that for trajectory τ_k, if we gave rewards at each time step, $R_{t,k}$, then we would have the following piecewise function

$$R_{t,k} = \begin{cases} 0 & 1 \leq t \leq T-1 \\ R_k & t = T \end{cases} \tag{5}$$

where R_k is the child algorithm's performance metric, after applying the transformations specified from τ_k. This is also called a *sparse* reward setup. Secondly, similar to the N-armed bandit problem [6], our model only depends on actions and rewrads. This means that we have a stateless reinforcement learning problem, With these modifications, our trajectory can be simplified to be the following:

$$\tau = \{a_1, a_2, \cdots, a_T\}$$

Notice that while R_k could be considered part of the trajectory, we will remove it for convenience and only consider the transformations $a_t, 1 \leq t \leq T$ to be the trajectory.

Now that we have defined a single trajectory τ, we can continue with the objective function. However we will notice that the reward signal, $R(\tau)$, is non-differentiable as it is solely a performance metric for our child algorithm. Despite this, we can perform a decomposition of our objective function such that we introduce differentiable weights. This can be seen in "Derivation of the Policy Gradient" [9].

Derivation of the Policy Gradient

$$
\begin{aligned}
J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\
\implies \nabla_\theta J(\theta) &= \nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \sum_{k=1}^{M} P(\tau_k|\theta)R(\tau_k) && \text{(expanding the expectation)} \\
&= \sum_{k=1}^{M} \nabla_\theta P(\tau_k|\theta)R(\tau_k) && \text{(linearity of gradient operator)} \\
&= \sum_{k=1}^{M} P(\tau_k|\theta)\nabla_\theta \log(P(\tau_k|\theta))R(\tau_k) && \text{(log-derivative trick)} \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t)R(\tau)\right] && \text{(collapsing into the expectation)}
\end{aligned}
$$

0.3.3.1 REINFORCE with baseline

From above, we have the following objective function (which is the vanilla policy-gradient method [9]),

$$J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t)R(\tau)\right] \tag{6}$$

While this is a good start, the variance for this objective function is quite high. In order to combat this, we can attempt to reduce the variance of the objective function and introduce a baseline function, b_s. Due to our stateless model, we can simplify b_s into $b \in \mathbb{R}$. With this implementation, we do not want to actually change the optimization process for the objective, and thus, need to make sure the gradient is unchanged. For this, we go to the "Unbiased Baseline Proof" [11]. It is important to note that the baseline function must be a function of the state (or a constant if stateless) in order for the estimator to be unbiased.

Unbiased Baseline Proof

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_t) R(\tau)\right] \\
&\overset{?}{=} \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_t)(R(\tau)-b)\right] \\
&= \nabla_\theta J(\theta) - \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_t) b\right] && \text{(by linearity of expectation operator)} \\
&= \nabla_\theta J(\theta) - \sum_{t=1}^{T}\mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t) b\right] && \text{(by linearity of expectation operator)} \\
&= \nabla_\theta J(\theta) - \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[b\sum_{a}\pi_\theta(a)\nabla_\theta \log \pi_\theta(a)\right] && \text{(for arbitrary } t\text{, expand R.V. } a_t\text{)} \\
&= \nabla_\theta J(\theta) - \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[b\sum_{a}\nabla_\theta \pi_\theta(a)\right] && \text{(inverse log-derivative trick)} \\
&= \nabla_\theta J(\theta) - \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[b\nabla_\theta \underbrace{\sum_{a}\pi_\theta(a)}_{1}\right] && \text{(by definition of a valid prob. dist.)} \\
&= \nabla_\theta J(\theta)
\end{aligned}
$$

Now that we have shown that adding a baseline does not affect our gradient, we are left with the form shown in Equation 7.

$$J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T}\log \pi_\theta(a_t)(R(\tau)-b)\right] \tag{7}$$

Now, what type of baseline should we choose? We know it has to be a constant as we have a stateless problem. Due to the baseline being subtracted from the reward, we could treat the difference as an advantage for that given trajectory. In other words, we could have b be some average of the trajectories sampled. This way, if a certain trajectory is below average, the advantage will be negative. Likewise, if it is above average, it will be positive. This then acts as a signal to the controller to either go in the opposite direction of the actions or follow similar actions in the gradient update. Thus, we define the advantage

$$A_k = R_k - b \tag{8}$$

While we could use a simple arithmetic mean, we will follow [3] and use the exponential moving average. With α being a hyperparameter and $1 \leq k \leq M$, we have

$$b_0 = 0 \tag{9}$$

$$b_{k+1} = \alpha b_k + (1 - \alpha) R_k \tag{10}$$

$$b = b_{M+1} \tag{11}$$

Lastly, in order to stay consistent with [4] and take into consideration the unbalanced datasets, we will instead opt to use *balanced accuracy* as the child algorithm's performance metric. This is a better metric for evaluating unbalanced datasets and is illustrated in Equation 12.

$$R_k = \frac{1}{2}\left(\underbrace{\left(\frac{TP}{TP + FN}\right)}_{\text{Recall}} + \underbrace{\left(\frac{TN}{TN + FP}\right)}_{\text{Specificity}}\right) \tag{12}$$

where T/F denotes "True/False" and P/N denotes "Positive/Negative," giving four distinct metrics.

0.3.3.2 Regularization

Learning rate: As with any gradient updating, we can choose the magnitude of our learning rate, γ. However, we do not necessarily need to choose a constant value. Instead, we can use what is called *learning rate scheduling*, which is the practice of changing your learning rate dynamically during the training process. It thus will be a function of our epoch w: γ_w. The reason for a dynamic learning rate is due to the high variety of the datasets. As each dataset is different, the dynamic learning rate allows one to explore the space more efficiently in the beginning due to the magnitude of the learning rate being larger. Then, as the magnitude decreases, it helps the model to converge as smaller gradient steps are performed. This can be seen in Figure 5.

Entropy: Other than using learning rates, one can use the concept of *information entropy* [8]. Inherently, entropy is a way to measure uncertainty in a distribution. Thus, we can attempt to maximize this uncertainly for our policy π_θ, leading to more exploration as it will prevent the

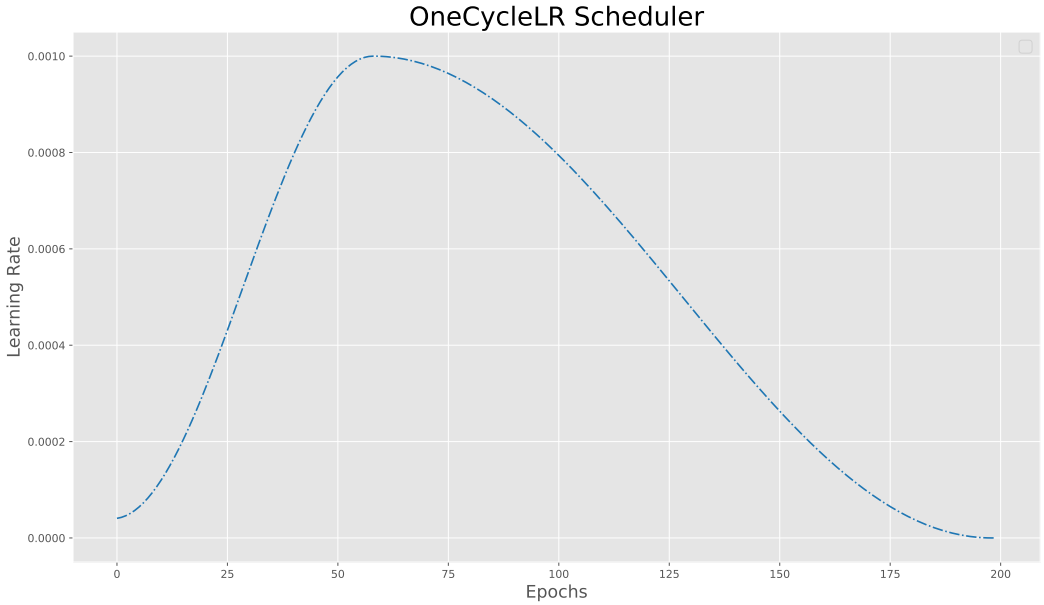

Figure 5: OneCycleLR with a maximum learning rate, 0.001, and maximum epoch 200.

model from converging onto a simple action. The formal definition of entropy is that if $\mathcal{X}$ is a random variable and is distributed according to $p : \mathcal{X} \mapsto [0, 1]$, define the entropy as:

$$H(\mathcal{X}) = -\sum_{t=1}^{T} P(x_t) \log P(x_t) \tag{13}$$

Increasing the entropy effectively smooths out distributions as $x_t \in \mathcal{X}$ entries with lower probabilities are considered more unlikely and carry a higher amount of meaningful information. This can be seen in Figure 6.

We wish to smooth out our policy, π_θ, to enforce policy exploration. Thus, our policy π_θ can be thought of as the random variable $\mathcal{X}$ in Equation 13. With the addition of the different trajectories, our entropy can be formulated as the following

$$H(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left[-\sum_{t=1}^{T} \pi_\theta(a_t; \tau) \log \pi_\theta(a_t; \tau) \right] \tag{14}$$

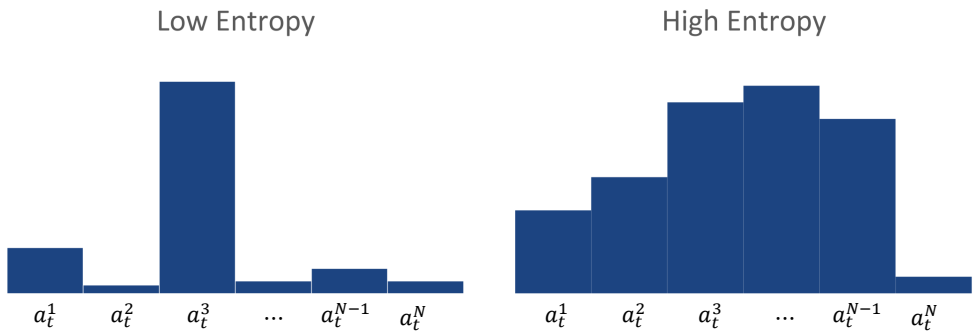

Figure 6: The difference between a low entropy and high entropy distribution

Inserting entropy into our objective function with weight ϵ gives us

$$J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T}\log\pi_\theta(a_t)(R(\tau)-b)\right] + \epsilon H(\pi_\theta) \tag{15}$$

$$= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=1}^{T}\log\pi_\theta(a_t)(R(\tau)-b) - \epsilon\pi_\theta(a_t;\tau)\,\log\pi_\theta(a_t;\tau)\right] \tag{16}$$

$$\propto \frac{1}{M}\sum_{k=1}^{M}\sum_{t=1}^{T}\log\pi_\theta(a_{tk})A_k - \epsilon\pi_\theta(a_{tk})\,\log\pi_\theta(a_{tk}) \tag{17}$$

where we choose to do an approximation to $\mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\cdot]$. This gives us our final form for the objective that we use in practice,

$$\boxed{J(\theta) = \frac{1}{M}\sum_{k=1}^{M}\sum_{t=1}^{T}\log\pi_\theta(a_{tk})A_k - \epsilon\pi_\theta(a_{tk})\,\log\pi_\theta(a_{tk})} \tag{18}$$

Early Stopping: Lastly, to further enforce exploration, we used early stopping in our optimization process. In short, during the training process, one keeps track of the best performing model parameters (based off of the mean validation accuracy of all trajectories for that given epoch).

0.3.4 Inference

After training, we have a network with trained parameters, θ, giving us the policy π_θ^*. Our goal is to then to sample from this network to get a final trajectory τ^* that we will use to transform our data and determine its test accuracy. Note that we have a total of N possible actions at each time step t. Thus, we will denote the specific action at time step t as $a_t^i,\ 1 \leq i \leq N$. Now to sample this trajectory, we have three options:

(1) Random Sampling: This is the default method (i.e. a forward pass our of model, such as was done during training. Although efficient and useful for exploration, it is not as stable as the other choices.

(2) Nucleus Sampling: Similar to random sampling, this is also just a forward pass of the model. However, the main difference is that at each time step t, we truncate the discrete probability distribution y_t. This effectively makes it more stable as it eliminates the unlikely tails of the distribution during sampling.

(3) Beam Search: Unlike the other two sampling methods, this acts more like an optimization scheme. Using the joint probability space spanned by the softmax layers, we can sample the best B actions at each time step, giving us multiple possible paths. This allows one to explore paths from π_θ^*.

0.3.4.1 Random Sampling

Given a trained policy π_θ^*, we can sample it to obtain τ^*. Referencing Figure 3., at each time step t, we have a distribution y_t. With the model in evaluation mode (not tracking gradients in our case), we simply sample at each time step: $a_t \sim y_t$. Doing this for all of the T steps, we obtain the trajectory $\tau^* = \{a_1, a_2, a_3, \cdots, a_T\}$. This method is quite simple but is susceptible to sampling sub-optimal actions and so is the last choice for our sampling method at inference.

0.3.4.2 Nucleus Sampling

As mentioned above, it is possible to sample sub-optimal actions. A concrete example of when this might happen would be if our distribution y_t has an action that has extremely low probability. Even though it has a low probability, we could still sample this action, causing our τ^* to give drastically worse results. To combat this, we attempt to eliminate the unlikely actions at each time step t. Specifically, we aim to pick the smallest set $V^{(p)}$ such that starting from the most likely actions in the distribution, we add actions until we have reached a specified sum p. Formally, at a fixed time time step t, our new sum p' is chosen

$$p' = \sum_{a_t^i \in V^{(p)}} P(a_t^i \mid a_t^{1:j-1}) \geq p \tag{19}$$

where $a_t^{1:j-1}$ denote the currently, existing actions in $V^{(p)}$. Next, we set all other actions to zero and normalize our discrete distribution so that it remains a valid probability distribution. Specifically our new discrete probability distribution is modeled as the following,

$$P'(a_t^i|a_t^{1:j-1}) = \begin{cases} P(a_t^i|a_t^{1:j-1})/p' & \text{if } a_t^i \in V^{(p)} \\ 0 & \text{otherwise} \end{cases} \tag{20}$$

This adaptation allows one to then sample like random sampling but at each time step, truncate the distribution and remove the highly unlikely actions. A visual representation of this process is depicted in Figure 7. You will notice that a_t^{N-1} and a_t^N are shaded differently. These represent the actions that were chosen to be set to 0. The other actions that sum to p' are a part of the set $V^{(p)}$. The resulting actions a_t that were sampling at each time step t from the truncated distributions are our trajectory $\tau^* = \{a_1, a_2, a_3, \cdots, a_T\}$.

0.3.4.3 Beam Search

Our final method of inference is Beam Search. For this method, instead of changing the distributions that we are sampling from, we aim to actually explore multiple trajectories across the whole policy π_θ^*. This effectively acts as an optimization scheme where we can then take the

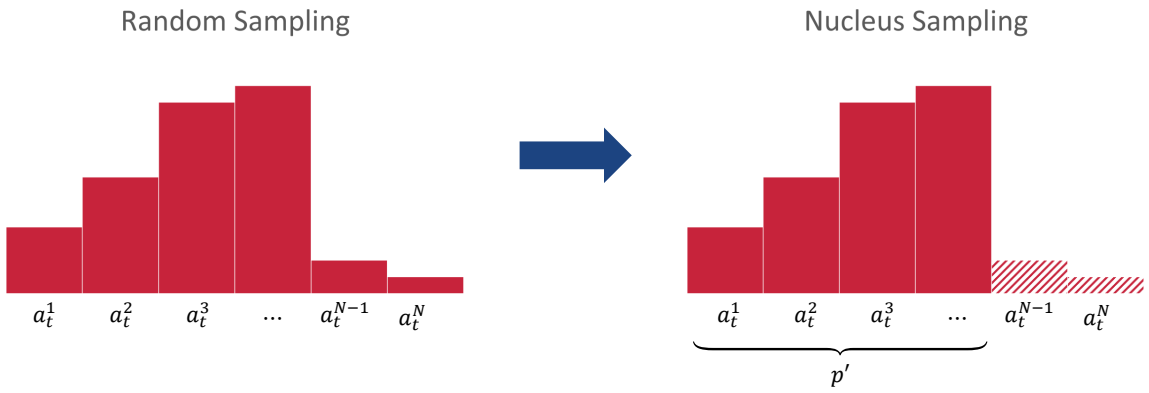

Figure 7: Random Sampling vs. Nucleus Sampling

best possibly trajectory(s) specified by our beam width B. As such, it combats the issue of not considering an ultimately better trajectory that went unnoticed due to choosing a more likely action at a previous time step t. Below we will make use of "The Conditional Probability Chain Rule."

The Conditional Probability Chain Rule

$$
\begin{aligned}
P(a_1, \cdots, a_T \mid x) &= P(a_T \mid a_1, \cdots, a_{T-1}, x) P(a_1, \cdots, a_{T-1} \mid x) \\
&= P(a_T \mid a_1, \cdots, a_{T-1}, x) P(a_{T-1} \mid a_1, \cdots, a_{T-2}, x) P(a_1, \cdots, a_{T-2} \mid x) \\
&\vdots \\
&\prod_{t=1}^{T} P(a_t \mid m_{t-1}, x) \qquad \text{(where } m_t = \{a_1, a_2, \cdots, a_t\},\ m_0 = \{\}\text{)}
\end{aligned}
$$

We can then consider our trained policy π_θ^* as a joint probability space. Using this space, we could test every possible trajectory, but if there are a lot of actions or time steps, this would

take too long. Beam search finds a balance by considering B possible actions at each time step and dynamically keeping track of the best sequences up to that time t. Specifically, using conditional probability chain rule. We have

$$\zeta_t = \prod_{i=1}^{t} P(a_i \mid m_{i-1}, x) \tag{21}$$

For instance, at the first time step, we find the top B actions with the highest scores $\zeta_1 = P(a_1 \mid a_0)$. Using these B actions, we can then create B instances of the next softmax layer $y_{j,2}$, $1 \leq j \leq B$, where j is the instance index. Now, across all instances, we then pick the top B actions with the highest score $\zeta_2 = P(a_2 \mid a_1, a_0) \cdot P(a_1 \mid a_0)$. Using these B actions, we create B instances of the next softmax layer $y_{j,3}$, $1 \leq j \leq B$. We continue this process across T time steps and end up with the top B trajectories that were found using a beam width B. Notice that we can also reduce beam search into two common inference strategies that are found in Table 2. The beam search's algorithm is stated as Algorithm 2. A visual example of beam search is depicted in Figure 8.

Algorithm 2: Beam Search

Input : B, π_θ^*
Output: τ^*
for each t **in range(T) do**
  ζ_t =P($a_1, a_2, \cdots, a_t \mid a_0$) ; `// compute scores across ALL instances`
  best_actions = torch.topk(ζ_t, B) ; `// find top` B `actions from` ζ_t
  for j, action in enumerate(best_actions) do
    $y_{j,t+1}$ = model.forward(action) ; `// create` B `instances of` y_{t+1}
  end
end
τ^* = torch.topk(ζ_T, 1) ; `// return highest score trajectory`

Value of B	Inference Strategy
1	Greedy
N	Exact

Table 2: Inference Strategies Adapted from Beam Search

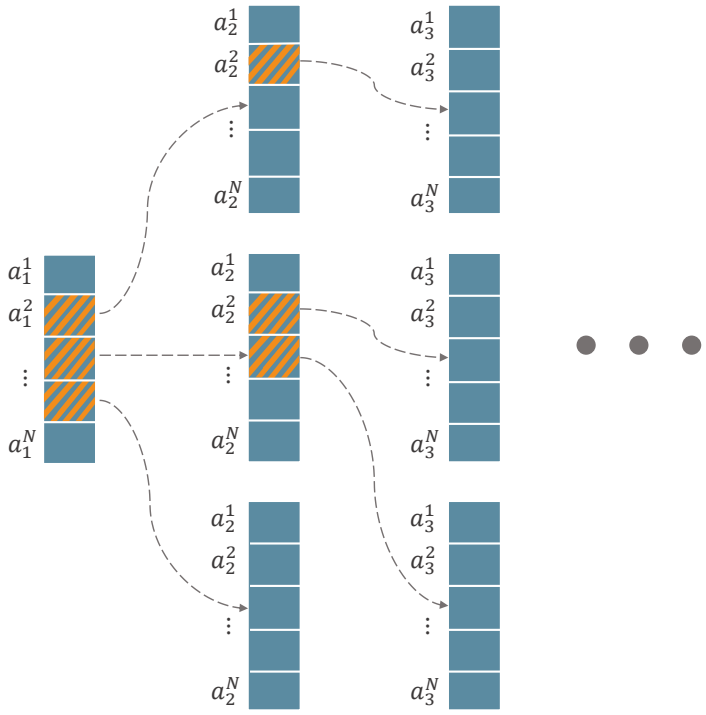

Figure 8: A Visual Illustration of Beam Search with $B = 3$.

0.4 Data Description

0.4.1 Source of Data

In order to keep complexity manageable for this project, we opted to start with simpler datasets. Inspired by [4], we used the OpenML-CC18 Curated Classification benchmark suite [2], the same 60 datasets, as in [4]. Nearly all of the datasets are non-synethetic and allow a realistic way to evaluate a given model as the structure of each dataset varies in its numerical features, symbolic features, number of features, and number of instances. Lastly, OpenML's website for the suite proved to be useful as a tool to see how other models performed on the datasets along with the pre-processing pipeline.

0.4.2 Data Exploration

We should first note that each of the dataset is tabular, meaning it can be formulated as a matrix $\mathbb{R}^{Q\times P}$, where Q is the number of instances and P is the number of features. In this section, we will not be looking into each and every set's structure but briefly looking at the decomposition of all the datasets. An example of what a single dataset looks like can be found in Figure 3, where the goal is to predict if the sample is tipped to the right, left, or balanced.

	left-weight	left-distance	right-weight	right-distance
0	1.0	1.0	1.0	1.0
1	1.0	1.0	1.0	2.0
2	1.0	1.0	1.0	3.0
3	1.0	1.0	1.0	4.0
4	1.0	1.0	1.0	5.0
⋮	⋮	⋮	⋮	⋮

Table 3: An Example Dataset from OpenMLCC-18

Data properties include (but are not limited to) the number of features, the number of categorical features, the number of features, and the number of instances. One metric we can look at is how much of the datasets are numerical versus categorical. Furthermore, we want to consider the number of features rather than the whole dataset as certain datasets might have mixed data types. Thus, we simply take it to be a fraction between the numerical/categorical features and the overall features.

The corresponding pie chart is shown in Figure 9. As you can see from the pie chart, a large majority of features are numerical ($\sim 85\%$). Again, this is across all features of all datasets, as it then considers datasets with mixed types. Although it is a large majority, there are still categorical features ($\sim 15\%$), so in order for our model to remain adaptable, we must not make any stipulations. Specifically, we could have our model always perform encoding as the first step; however, then our pipeline would not maintain adaptability. Therefore, we make no assumptions and let the model figure out what to do. Next, it is important to look at the number of features and instances. In the box plots shown in Figure 10., we can see that the number of instances are

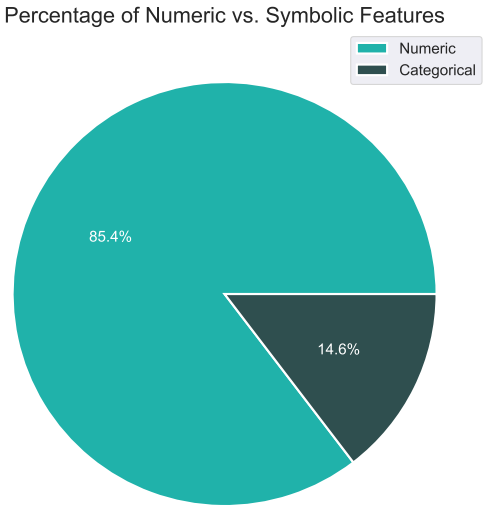

Figure 9: OpenML-CC18 Benchmark Suite: Percent of categorical vs. numerical features.

closely concentrated to be below 10,000 instances. However, we can also see that there are quite a few outliers that go all the way up to 100,000 instances. Similarly for the number of features; while most of the number of features are closely concentrated to be below 100, there are some datasets that go all the way past 800 features. There is a high variety of datasets in this suite that we used and there is a mix of numerical as well as categorical datasets. Due to the diversity, it will be a good test for our approach to see if it can perform well across various types of datasets. These results will be discussed in Section 0.5.

0.5 Results

0.5.1 Implementation Setup

Computer Hardware: For our setup, we chose to not use distributed computing but to use a single computer's computing power. For each classifier, other than Naive-Bayes, the *n_jobs*

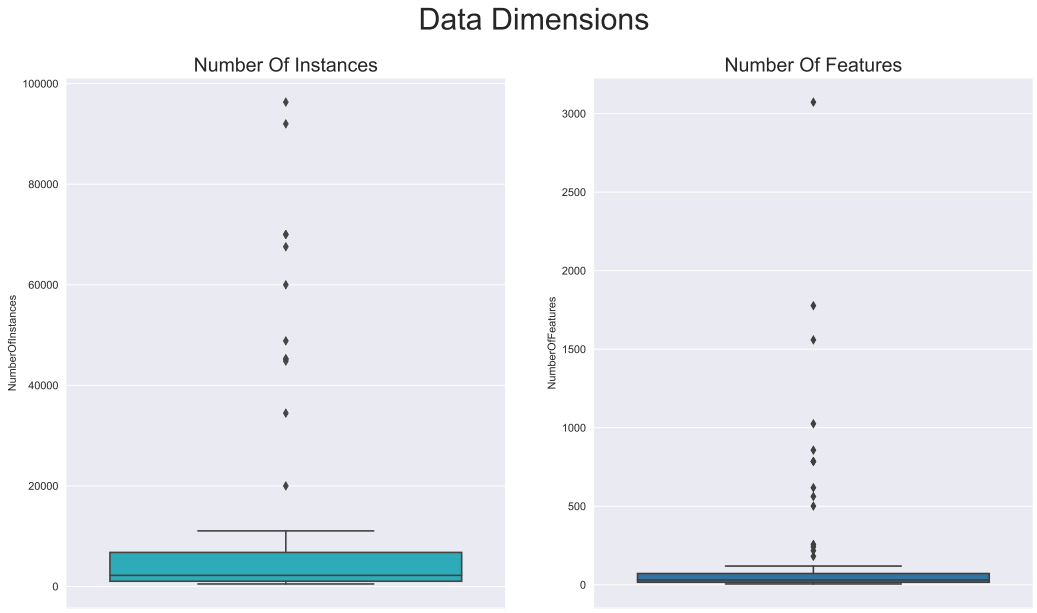

Figure 10: OpenML-CC18 Benchmark Suite: Number of instances and features.

parameter was set to be -1 so that all cores were used. Additionally, the controller network as well as the data was cast to our GPU device during training. Table 4 provides more specifics on our setup.

Hardware	Software	Hyperparameters
i7-8750h CPU (6 cores) GTX 1060 Max-Q	n_jobs (Pipeline/Classifier)	M = 100 epochs = 200 $\alpha = 0.95$ $\gamma = 0.001$ $\epsilon = 0.001$

Table 4: Implementation Details

Hyperparameters: For the hyperparameters of the learning rate and entropy, the optimal values were hand-tuned. The other hyperparameters such as the number of epochs, number of units for an RNN cell, and embedding layer were chosen arbitrarily in the beginning and were not altered with the hope that they were as general as possible. In our case, the best maximum

learning rate was found to be $\gamma = 0.001$ while the best value of entropy weight was found to be $\epsilon = 0.001$. Finally, the weight in the updating of the baseline was set to be $a = 0.95$, matching the setup of [12]. For the iteration parameters, we used $M = 100$ for the number of episodes and 200 epochs.

0.5.2 Results

0.5.2.1 Raw Results

Firstly, we will look at how well our model does across all the datasets. With the hyperparameters and action space fixed, we run the model on every $(\mathcal{X}, f(\cdot), \mathcal{A})$ pair, where $f(\cdot)$ is the Naive Bayes algorithm. In order to determine if our trajectory τ^* is effective, we get a baseline test accuracy before and after applying the transformatons specified from τ^*. We define the following trajectory to be the baseline trajectory,

$$\tau_0 = [\text{Ordinal Encoding}, \text{Identity Map}, \cdots, \text{Identity Map}] \tag{22}$$

where Ordinal Encoding is applied on a column-wise basis. Therefore, if a dataset has completely numerical features, then Ordinal Encoding turns into the Identity map as nothing is encoded. Using this baseline trajectory, we can then obtain a test accuracy on our dataset before training starts, giving us $Acc\ (\tau_0, a)$. Then, after training is completed, we can apply our trajectory τ^* to the test set and obtain a new test accuracy, $Acc\ (\tau^*, a)$. Note that our child algorithm hyperparameters' a are not optimized and are set to be the default values. With these two test accuracies, we can then determine how well our model did in improving the performance across all datasets. This is evaluated using a relative improvement metric,

$$\delta_{\text{our\_model}} = \delta_{our} = \frac{Acc(\tau^*, a) - Acc(\tau_0, a)}{Acc(\tau_0, a)} \tag{23}$$

With this metric, we can compute a value for each dataset. Looking at all the values as a distribution, we can then look at the results in Table 5. Note that we sampled the trained policy π^*_θ using the three different inference methods covered in Section 0.3.4. Moreover, for beam search we used

	δ_{our} (Random)	δ_{our} (Nucleus)	δ_{our} (Beam Search)
mean	6.69%	7.62%	**11.41**%
std	37.03%	35.29%	**18.67**%

Table 5: Our model's improvement

$B = N$ as our action space is relatively small. In Table 5, we can see that, on average, Beam Search did the best by improving our dataset's accuracy with $\sim 11.5\%$ relative improvement score. We can also see that its standard deviation was the lowest at $\sim 18.5\%$, suggesting that it was a more stable option compared to the other inference methods. Moreover, we do see an improvement from random sampling to nucleus sampling in both the mean as well as the standard deviation, albeit not as large of a jump from nucleus sampling to beam search.

Next, we can inspect how each individual dataset performed. This can be seen by using a bar chart and having the y-axis represent the relative improvement score, δ_{our}, and the x-axis represent the dataset's ID number. This is seen in Figure 11.

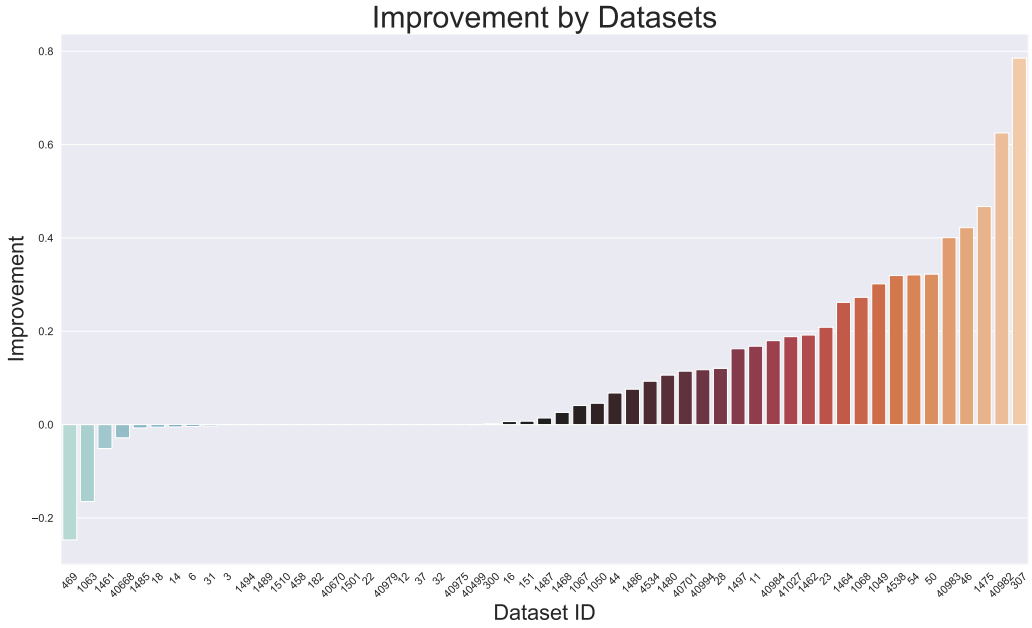

Figure 11: A Visual Showing Results by Dataset ID

Looking at Figure 11., we can see that a large majority of datasets had some increase in performance. However, there are some datasets whose improvement did not increase at all and some that even got worse. Lastly, although our model was supposed to find a better trajectory than the baseline, there were some datasets in which the found trajectory τ^* actually caused worse performance. In practice, there could be a simple fix by checking if our baseline test accuracy is higher than our test accuracy with the found trajectory, however, this in theory should not happen and most likely points to an optimization fault. More specifically, we could have gotten stuck in a local optimum, resulting in sub-optimal performance. The other possible reason is if our training/validation/test split was just too difficult to learn from. One way to combat this would be to use cross validation; however, this does result in significantly slower training.

One hyperparameter that is quite important to consider is the number of trajectories sampled for epoch: M. Although our implementation ended up with $M = 100$ to be the best value, it was considerly slower than using $M = 1$. Specifically to do 50 of the datasets and Naive Bayes, it took

$\sim$ 49 hours with $M = 100$ and $\sim$ 1 hour with $M = 1$. Multiprocessing was not used on the different trajectories sampled. Due to the performance increase, the slower training time is worthwhile; however, one might consider trying to make the model more sample-efficient so that it is much faster but still yields competitive results. For a one-sided comparison, we can look at Figure 12. to see how much more of an improvement is obtained when using $M = 100$ versus $M = 1$ (i.e. $\delta_{our}(M = 100)$ - $\delta_{our}(M = 1)$).

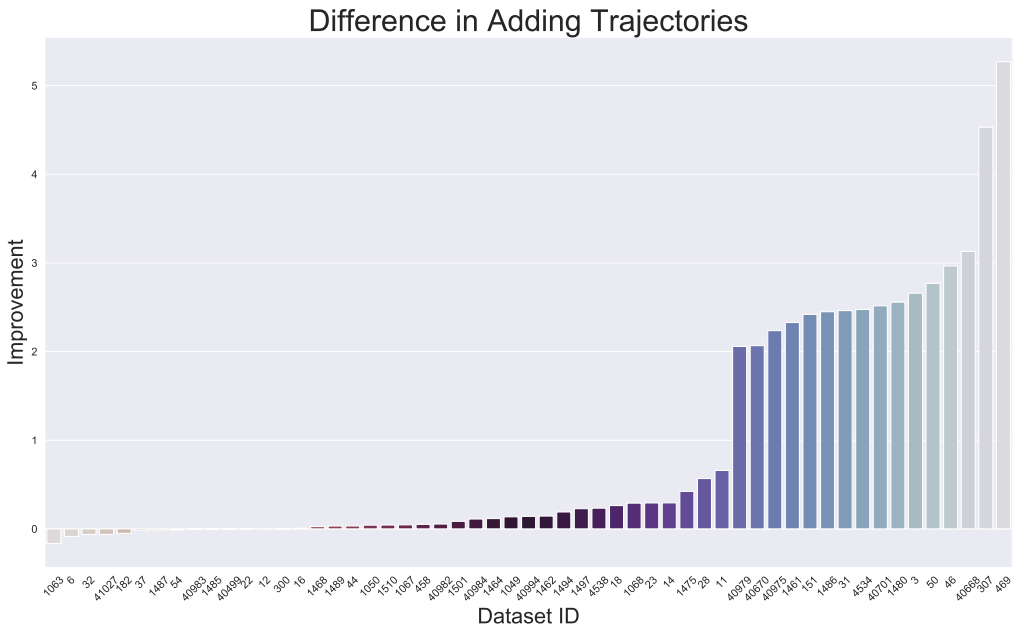

Figure 12: Comparing δ_{our} for $M = 100$ and $M = 1$

0.5.2.2 Relative Results

Next we compare to the results in [4]. A preview of their results are shown in Table 6.

In Table 6., for each dataset in the OpenML-CC18 benchmark suite, they tested three different child algorithms: K-Nearest Neighbors (KNN), Naive Bayes (NB), and Random Forest (RF). Here, d_effective is their equivalent to our τ^*, as is their d to our τ_0. Likewise, a are the hyperparameters

did	Algorithm	Acc (d_effective,a*)	Acc (d,a*)	Acc (d,a)
3	knn	87.21	80.12	79.22
3	nb	78.05	76.08	59.17
3	rf	97.98	97.95	94.74
6	knn	96.27	96.22	95.56
6	nb	65.48	64.10	64.10
⋮	⋮	⋮	⋮	⋮

Table 6: Effective pre-processing for AutoML Results

of the child algorithm. Lastly, for our results, we used the same datasets as they did and chose NB as our child algorithm to test it on. Now, to compare their results to ours, we must consider two main differences in how the results were produced:

(1) Different training/validation/test splits were used

(2) The child algorithm's hyperparameters were optimized: $a \rightarrow a^*$

As in Section 0.5.2.1, we can determine how much of an improvement was created using their pre-processing pipeline. Despite them possibly having different data splits, we can use the same method. Letting d_effective $\rightarrow \tau^*$ and d $\rightarrow \tau_0$, we define their metric of improvement

$$\delta_{\text{baseline\_paper}} = \delta_{bs} = \frac{Acc\ (\tau^*, a^*) - Acc\ (\tau_0, a^*)}{Acc\ (\tau_0, a^*)} \tag{24}$$

Next we can compare their improvement to our improvement, and thus, get a value for each dataset. This is formally done by taking,

$$\Delta_{diff} = \delta_{our} - \delta_{bs} \tag{25}$$

In Table 7., we can compare the mean and standard deviation of our method to that of the baseline paper. In general, our method performs slightly worse than theirs; however, it is centered around 0. The main reason for this is that we did not optimize the pre-processing parameters and only used the default parameters for each action in the action space. Despite this disadvantage, it mostly produced similar results when using beam search. While beam search was the best option, all standard deviations were similar. For the raw improvement, Table 5 and 7 show that most datasets had an improvement anywhere from 0.0 and 0.2 (that is roughly 0% to 20%). Although some

	Δ_{diff} (Random)	Δ_{diff} (Nucleus)	Δ_{diff} (Beam Search)
mean	-2.08%	-0.90%	**-0.86**%
std	17.82%	17.61%	**17.44**%

Table 7: How our model compares to the baseline paper

datasets actually got worse with a score of -0.2, many more datasets actually had an improvement value of 0.8. Likewise, when compared to the baseline paper, we see that Δ_{diff}(Beam Search) is generally normally distribution about 0.0. Again, it also has some datasets that have a lower improvement value near -0.3, but we also see that we have datasets that had an improvement value of 0.4 up to 0.8. This generally shows a normal distribution and shows that our method is comparable to theirs.

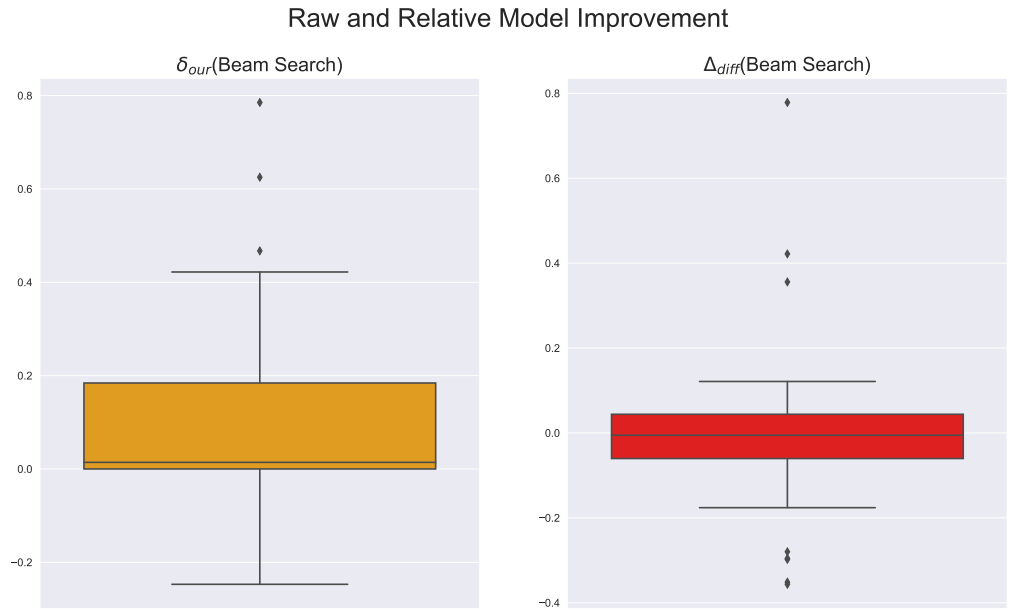

Figure 13: Raw and Relative Results

0.5.3 Different Sampling Methods

Moving on from the quantitative results, we can look into the sampling methods that were chosen for each time step. For this, we will simply group the transformations at each time step (across all datasets) into their respective category: rebalancing (R), normalization (N), discretization (D), feature engineering (F), imputation (I), encoding (E), and *None*. For instance, across all datasets used, we suspect that encoding should be used in the first step. We will visualize this using density heatmaps and look at each inference method separately. Each column adds up to 100% as it counts how many of that type of transformation was chosen in that given time step (e.g. 20% for normalization on $t = 1$ means that 20% of the transformations were of normalization type for the first time step).

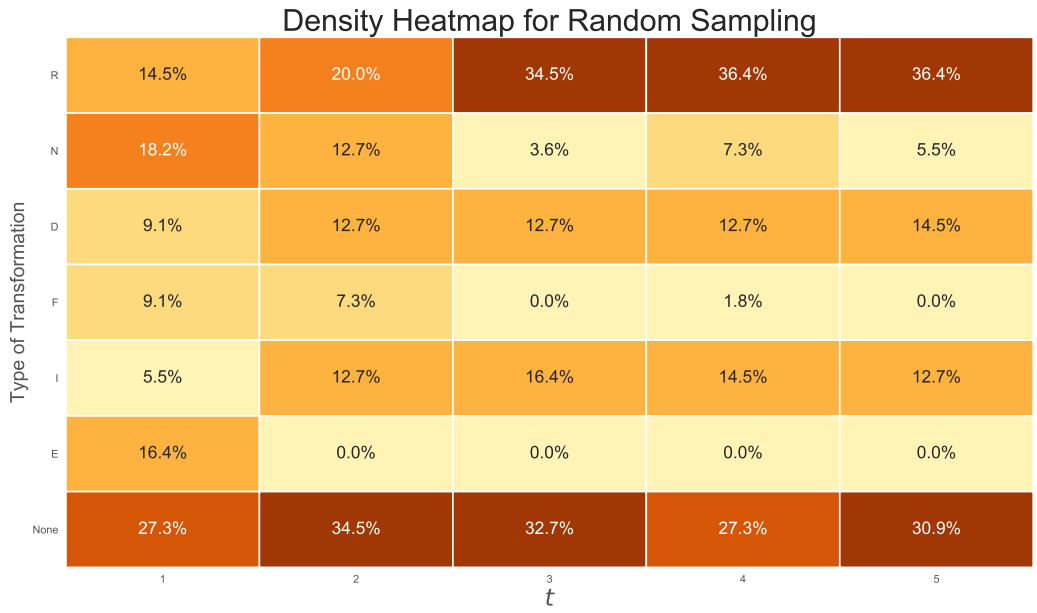

Figure 14: Density Heatmap: Random Sampling

The random sampling heatmap, 14, shows that the most common type of transformation for

the first time step at $t = 1$ was *None* ($\sim 27\%$). Normalization ($\sim 18\%$) and encoding ($\sim 16\%$) are next. This makes sense, as some of the datasets were categorical and would need to be encoded first. Other interesting trends observed include that rebalancing seems to increase over time, normalization seems to decrease over time, and feature engineering decreases over time. It also appears that discretization and imputation seem relatively constant after the first time step. This could suggest that the the time step at which imputation or discretization are applied are not very important. The high density for rebalancing and the identity map could suggest that either more data is needed to gain insights into the dataset (rebalancing) or no action should be applied to combat overfitting.

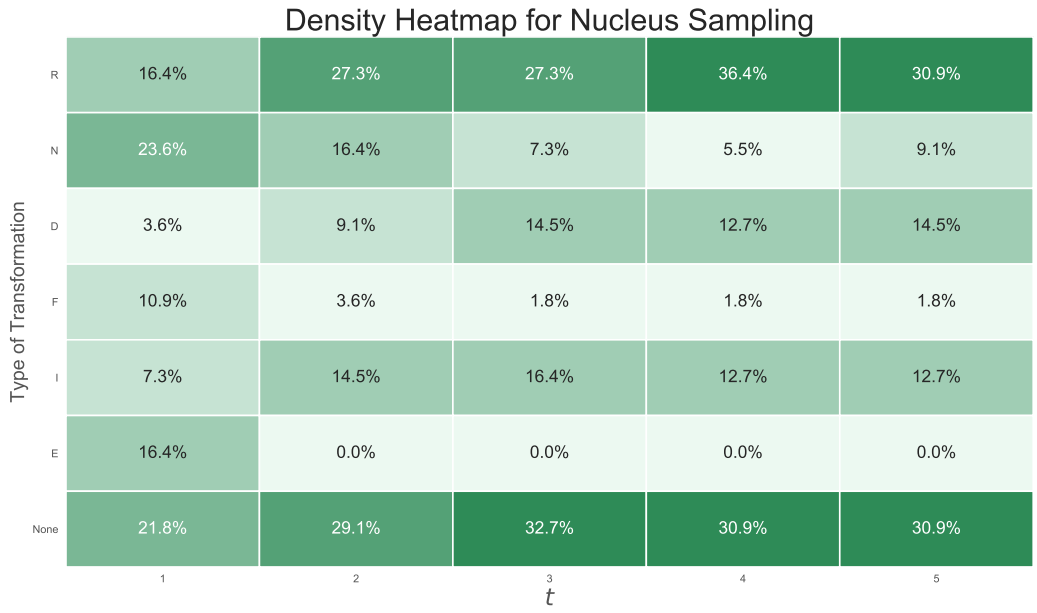

Figure 15: Density Heatmap: Nucleus Sampling

For nucleus sampling, normalization was overall chosen more than *None* as the first type of transformation as opposed to when we used random sampling for the inference method. Additionally, *None* was chosen as the most likely transformation on the third time step as well. Other

than these main differences, the general trends are still seen over time: normalization decreases, feature engineering decreases, and discretization/imputation stay mostly constant after the first time step. Recall that although the differences seem small, the quantitative results do suggest that using nucleus sampling over random sampling is advantageous.

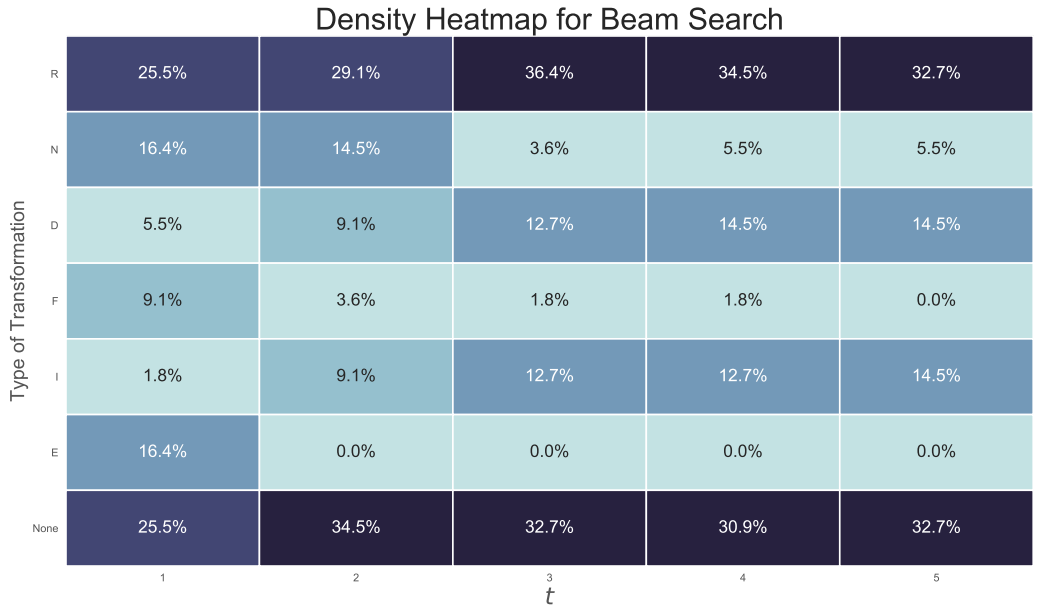

Figure 16: Density Heatmap: Beam Search

For beam search, we can see that rebalancing ($\sim 26\%$) and *None* ($\sim 26\%$) are the most likely choices for $t = 1$. Like random sampling, the most likely step for $t = 3$ is also *None*. While the trends still hold for our beam search method, there are a few differences: normalization is most common for the first two time steps, and discretization/imputation are less likely for the first two time steps.

Although each inference method has slightly different patterns and produces different qualitative results, the overall narrative that the heatmaps provide is consistent. For smaller tabular datasets, rebalancing could be a way to improve overall accuracy even if it is not needed; normal-

ization is typically applied earlier on ($t = 1, 2$); encoding is done at $t = 1$; feature engineering is done early as well ($t = 1$); and discretization/imputation generally do not have a preference on where they are applied once the data is encoded/normalized ($t = 3, 4, 5$). Lastly, *None* is common across all t values, which most likely just point to how each dataset is different in the amount of transformations it needs.

0.6 Discussion

Future Work: Some immediate future advancements could be to implement a conditional aspect to our model such that the parameters are shared over datasets. This could possibly allow one to reduce the number of trajectories (M) required and still yield competitive results. Likewise, we could augment our action space to include parameters for both our pre-processing transformations as well as the child algorithm. With these two adaptations together, we could get better results at a faster rate. Separately, an application of this model could be to use the results to explore how transformations affect the manifold structure of a dataset at each time step. Ideally, one can run this model and get experimental evidence and then use manifold learning to study the theoretical aspect and postulate how to preprocess a dataset solely based off its manifold structure. It would also be interesting to explore this model and see which types of datasets it performs well on. This could contribute to the field of "Explainable AutoML," which to the best of our knowledge, has not been covered much yet.

Multiple Datasets and Hyperparameters: In general, the goal of any type of AutoML model is to take the user out of the equation. While the user is clearly not a prominent figure in our model, it did require the user to tune the learning rate value, epochs, and entropy weight. In practice, these values were chosen anecdotally and then fixed across all datasets when running results. However, an ideal model would be able to adapt to each dataset and set the parameters automatically. In this case, a conditional version of the model seems more appealing. As such, one should consider this flaw in the model as it takes away from the adaptability of our model, since we found the hyperparameters specific to the dataset suite and child algorithm.

0.7 Conclusion

In this work, we demonstrated that Neural Architecture Search (NAS) can be applied to the data pre-processing part of a machine learning pipeline. Taking inspiration from two recent papers ([1] and [4]), we sought to make a contribution where our model was 1) not fixed in advance and 2) did not depend on transformation specific rules or empirical calculations. In this way, we were able to construct an *open* pipeline structure that is scalable to any type of tabular dataset or application. We chose Naive Bayes (NB) as our child algorithm and were able to get comparable results to a baseline paper ([4]) despite not optimizing the parameters of the pre-processing transformations. We hope that we can augment this model to include a conditional aspect such that the training can become more efficient as well as include the parameters for the pre-processing transformations. With these extensions, we believe this model can be used as a tool to study the theoretical affects of pre-processing transformations on an arbitrary dataset. Although the concept of pre-processing is both data and model dependent, this might shed light on which manifold data structures yield the best results in a machine learning pipeline.

Bibliography

[1] Laure Berti-Equille. Learn2clean: Optimizing the sequence of tasks for web data preparation. In The World Wide Web Conference, WWW '19, page 2580–2586, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Bernd Bischl, Giuseppe Casalicchio, Matthias Feurer, Frank Hutter, Michel Lang, Rafael G. Mantovani, Jan N. van Rijn, and Joaquin Vanschoren. Openml benchmarking suites. arXiv:1708.03731v2 [stat.ML], 2019.

[3] Ekin Dogus Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation policies from data. CoRR, abs/1805.09501, 2018.

[4] Joseph Giovanelli, Besim Bilalli, and Alberto Abelló. Effective data pre-processing for automl. 05 2021.

[5] Bardenet R Bilenko M Guyon I Kegl B Hutter F, Caruana R and Larochelle H. Automl 2014 workshop @ icml. AutoML 2014 @ ICML, 2014.

[6] Michael N. Katehakis and Arthur F. Veinott. The multi-armed bandit problem: Decomposition and computation. Math. Oper. Res., 12:262–268, 1987.

[7] Stuart J Russell and Peter Norvig. Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,, 2016.

[8] Claude Elwood Shannon. A mathematical theory of communication. The Bell System Technical Journal, 27:379–423, 1948.

[9] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. The MIT Press, second edition, 2018.

[10] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In Proceedings of the 12th International Conference on Neural Information Processing Systems, NIPS'99, page 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[11] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning, 8:229–256, 1992.

[12] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. CoRR, abs/1707.07012, 2017.