Incorporating Automated Feature Engineering Routines into Automated Machine Learning Pipelines

by

Wesley Runnels

Submitted to the Department of Electrical Engineering
and Computer Science
in partial fulfillment of the requirements for the degree of Master of
Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 2020

© Massachusetts Institute of Technology 2020. All rights reserved.

Author . . . .

Wesley Runnels
Department of Electrical Engineering
and Computer Science
May 12th, 2020

Certified by . . . .

Tim Kraska
Department of Electrical Engineering
and Computer Science
Thesis Supervisor

Accepted by . . . .

Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Incorporating Automated Feature Engineering Routines into Automated Machine Learning Pipelines

by

Wesley Runnels

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2020, in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Automating the construction of consistently high-performing machine learning pipelines has remained difficult for researchers, especially given the domain knowledge and expertise often necessary for achieving optimal performance on a given dataset. In particular, the task of feature engineering, a key step in achieving high performance for machine learning tasks, is still mostly performed manually by experienced data scientists. In this thesis, building upon the results of prior work in this domain, we present a tool, `rl_feature_eng`, which automatically generates promising features for an arbitrary dataset. In particular, this tool is specifically adapted to the requirements of augmenting a more general auto-ML framework. We discuss the performance of this tool in a series of experiments highlighting the various options available for use, and finally discuss its performance when used in conjunction with Alpine Meadow, a general auto-ML package.

Thesis Supervisor: Tim Kraska
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

I would like to thank Professor Tim Kraska and the Kraska lab for taking me on as a research assistant and introducing me to the ideas behind this work. In particular, I would like to thank Zeyuan Shang and Emanuel Zgraggen for all their novel ideas and help with implementation details.

Contents

1 Introduction **14**

2 Prior and Related Work **16**

3 Feature Engineering Routine **18**

3.1 General Operation 18

3.2 Operation Details 19

3.3 Learning Algorithm 22

4 Algorithm Modifications and New Features **24**

4.1 Scoring Modification 24

4.2 Automatic Feature Selection 25

4.3 Budgeting by Time 27

4.4 The `cherrypick` Operation 27

4.5 Runtime Estimation 28

4.6 Model Selection 30

5 Using `rl_feature_eng` **32**

5.1 Training 32

5.2 Bulk Testing 33

5.3 Direct Testing 33

5.4 Incorporation into Alpine Meadow 34

6 Experimental Results **37**
6.1 Experimental Setup and Results 37
6.2 Analysis of Results 41

7 Extensions and Future Work **44**

8 Conclusion **46**

A Operation Types **49**

B Settings and Parameters **52**
B.1 Operational Settings 52
B.2 Algorithm Settings 53
B.3 Performance Settings 55
B.4 Output Settings 56

C Package Structure **57**

D Preprocessing **59**

E Feature Filtering **61**
E.1 Inverse Pairs 61
E.2 Highly Correlated Features 61
E.3 Redundant Operations 62
E.4 Constant Features 62
E.5 High Cardinality Categorical Features 62

List of Figures

3.1 Example Graph with 6 Nodes 20

4.1 Example Graph with 12 Nodes and `cherrypick` 28

6.1 Test score heatmap 42

List of Tables

3.1 Feature Growth with Composed Operations 21

6.1 Trial Settings 38
6.2 Validation Results 39
6.3 Testing Results 41

Chapter 1

Introduction

Despite the many recent advances in automated machine learning methods, one aspect remains particularly elusive to perform well: automated feature engineering. Indeed, feature engineering is often thought of as more of an art than a science. While plenty of sophisticated modeling techniques exist for extracting as much information as possible from a dataset, choosing among the innumerable possibilities when it comes to generating features and augmenting a dataset is a particularly difficult task for algorithms to handle, given the creativity and domain knowledge often required to efficiently and effectively select among the infinite possibilities. Thus, the task of feature engineering, among the most fundamentally important steps for maximizing modeling performance, has traditionally been left to manual experimentation and human intuition.

Recently, we have seen significant advancements in the domain of automated feature engineering which hold much promise in automating significant portions of what has traditionally been left to manual effort. Some preliminary works, which we discuss in the next section, introduce various methods of identifying a space of potential features and exploring it intelligently. These methods generally work by identifying a set of simple operations (e.g. addition, date extraction, logarithm) and exploring some subspace of features generated by applying these operations to the original features. If model quality can be improved with the addition of such

features, then by exploring the feature space in an intelligent way, the algorithms should be able to identify useful new features. Even though the proportion of useful features among all those explored may be small, by continually testing a large number of features, an algorithm may be able to compensate sufficiently for its relative lack of intuition and domain knowledge.

A further issue regarding the employment of these novel techniques is the incorporation of such routines into more complete auto-ML pipelines. While combining an automatic feature engineering routine with a simple, standard machine learning model (e.g. a generic random forest implementation) can produce promising results, it is a more complicated task to seamlessly incorporate such a routine into a comprehensive auto-ML framework. Among the many potential issues one may encounter are large increases in modeling time due to feature expansion and vulnerability to overfitting due to significant expansion of the feature space.

In this work, we describe `rl_feature_eng`, a Python package containing our implementation of a prior automated feature algorithm along with numerous modifications and additions that improve its utility in the context of a more general automated machine learning framework. Next, we describe the process of combining this feature engineering algorithm implementation with Alpine Meadow, a high-performing automated machine learning framework that lacks feature engineering capabilities. We then evaluate and compare the performance of `rl_feature_eng`, Alpine Meadow, and various combinations of them over a large corpus of datasets.

Chapter 2

Prior and Related Work

With the challenges of constructing auto-ML pipelines in mind, several auto-ML frameworks have been developed. As an example, a popular tool for this is the auto-sklearn library in Python [2] (Pedregosa et al. 2011). Upon encountering a new dataset, auto-sklearn leverages its experience with previous datasets to choose promising combinations of its numerous data preprocessing methods and classifiers to try.

A recent approach which we build upon in this work is the Alpine Meadow machine learning tool [3] (Shang et al. 2019), a framework achieving state-of-the-art performance in auto-ML competitions, such as the Data Driven Discovery (D3M) program held by DARPA. Alpine Meadow further explores the idea of identifying promising machine learning pipelines by referencing experience gained from exposure to prior datasets. Alpine Meadow, forming one component of the NorthStar utility, heavily prioritizes interactivity with the user and rapidly and frequently provides feedback on current model performance. This focus on interactivity, feedback, and interpretability makes it particularly appealing to users who lack the expertise of professional data scientists.

The field of automated feature engineering in particular remains a relatively unexplored area, with plenty of opportunity available for developing novel methods. Recently, there have been a few notable results in this space. One such effort is

the Deep Feature Synthesis (DFS) algorithm [4] (Kanter, Veeramachaneni 2015), which applies a set of transformations to all available features and selects the most promising among them. DFS served as a proof of concept of the idea of automated feature engineering, outperforming two-thirds of human teams it competed against in machine learning competitions in 2015.

More recently, several methods have been explored which use tree-based exploration methods to construct compositions of feature transformations. These methods aim to leverage efficient exploration algorithms to avoid the computational overhead of previous exhaustive search methods, while allowing the composition of several types of feature transformations. Among these is [1] (Khurana, Samulowitz, Turaga 2018), a framework which employs a reinforcement learning algorithm to guide exploration of feature generation. Particularly impressive is its combination of speed, performance, and feature intelligibility.

To this end, the core algorithm in `rl_feature_eng` is an independent implementation of the algorithm in [1], with some minor differences. On top of this base algorithm are a suite of improvements and adjustments that are intended to facilitate the algorithm's incorporation into a larger, more general auto-ML system. In the following sections, we describe these changes and the reasoning behind their inclusion.

Chapter 3

Feature Engineering Routine

In this chapter, we give an overview of the structure and function of the feature engineering routine upon which the `rl_feature_eng` package is based. This structure is largely based upon the work in [1]. The modifications to this base structure included in `rl_feature_eng` will be addressed in the subsequent chapter.

3.1 General Operation

Given a dataset, our goal is to discover new features that improve modeling performance. In `rl_feature_eng`, we only consider features that are transformations of features in the original dataset. For example, given a dataset with features `height`, `weight`, and `age`, both $\frac{\texttt{height}}{\texttt{weight}}$ and $\log(\texttt{age})$, among others, are valid features we might consider.

As in [1], `rl_feature_eng` is driven by a reinforcement learning algorithm that seeks to efficiently search for useful features to generate. Feature generation is done through the creation of `Nodes` on a `Graph` of transformations, where each `Node` represents a new, augmented dataset yielded by the application of the transformation to its parent(s). While [1] uses the term "transformation" to represent the generation (or removal) of new features through some function, we use the terms "transformation" and "operation" interchangeably. Indeed, in `rl_feature_eng` transformations are represented by `Operation` objects. The full list of operations currently available

in `rl_feature_eng` is available in Appendix A.

At the beginning of the feature engineering process, a `Graph` is created with a single root `Node` corresponding to the original dataset. The next step of the algorithm is to select an `Operation` to use on the root `Node` to yield a new feature set. The logic behind the selection of this `Operation` is governed by the reinforcement learning algorithm, which will be described later.

Once this `Node` is generated, its performance is evaluated and execution continues. The default performance metric for classification problems is the Macro F1 score for classification problems and (1 - relative absolute error) for regression problems, both taken over 5-fold stratified cross validation. The reinforcement learning algorithm, having observed the performance of the previous `Node`, selects another `Operation` to be performed on the `Node` of its choice: either the root or its child. This process repeats until the algorithm reaches a termination condition: for example, after having explored and evaluated 100 `Nodes`.

Figure 3.1 shows a possible `Graph` that could be constructed during algorithm operation for a given dataset. The numbers within the `Nodes` correspond to the order in which they were added to the `Graph`.

3.2 Operation Details

Upon selecting an `Operation` to use on a specific `Node`, the `Operation` is applied to all valid targets in the `Node`'s set of features. Note that both originally included and newly generated features can be valid targets. For example, a `log` operation, which acts upon numeric features, would be applied to all numeric type features in the `Node`, rather than to only a specific subset of them. In most cases, the application of an `Operation` only adds news features to the `Node` it acts upon. Some `Operations` (e.g. `union`) only combine features between nodes, while a select few (e.g. `feature_selection`) potentially remove features. The precise behavior of all `Operations` is described in Appendix A.

There are three feature types supported by `rl_feature_eng`: `numeric` (for standard numeric types), `categorical` (for e.g. string feature types), and `datetime`.

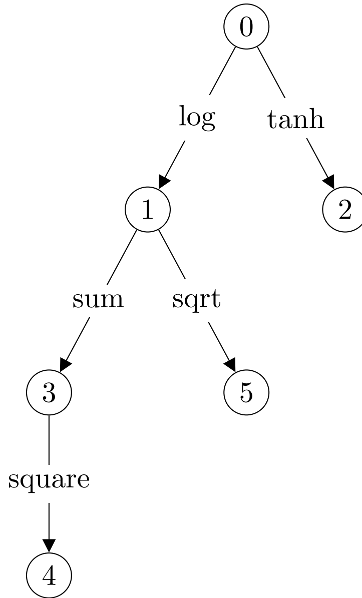

Figure 3.1: Example Graph with 6 Nodes

Nodes are labeled in chronological order of addition to the graph.

`datetime` type features can be passed in directly or inferred by `rl_feature_eng` if their name contains the string `"date"`; consult Appendix D for details.

To avoid redundancy, duplicate features are automatically omitted during feature generation. Other types of redundant and uninformative features, like constant features, are also removed automatically during feature generation. The details of this behavior can be found in Appendix E.

Since an `Operation` (excluding those which remove features, such as `feature_selection`) typically generates a new feature for each valid target in the targeted dataframe, dataset size can grow exponentially. For example, consider a base dataset with three numeric features `A`, `B`, and `C`, in conjunction with the `Graph` from Figure 3.1. Table 3.1 shows the features generated along the path to `Node` 4, as a result of applying `log`, `sum`, and `square`, in order, to the dataset.

Note that by default, `sum` does not generate $6^2 = 36$ new features, but instead $\binom{6}{2} = 15$; equivalent pairs such as `sum(A, B)` and `sum(B, A)` only yield one feature, while features such as `sum(A + A)` are omitted completely.

Node	Operation	Additional Features on This Step	Total Features
0	-	`A`, `B`, `C`	3
1	`log`	`log(A)`, `log(B)`, `log(C)`	6
3	`sum`	`sum(A, B)`, `sum(A, C)`, `sum(A, log(A))`, `sum(A, log(B))`, `sum(A, log(C))`, `sum(B, C)`, `sum(B, log(A))`, `sum(B, log(B))`, `sum(B, log(C))`, `sum(C, log(A))`, `sum(C, log(B))`, `sum(C, log(C))`, `sum(log(A), log(B))`, `sum(log(A), log(C))`, `sum(log(B), log(C))`	21
4	`square`	`square(A)`, `square(B)`, `square(C)`, `square(log(A))`, `square(log(B))`, `square(log(C))`, `square(sum(A, B))`, `square(sum(A, C))`, `square(sum(A, log(A)))`, `square(sum(A, log(B)))`, `square(sum(A, log(C)))`, `square(sum(B, C))`, `square(sum(B, log(A)))`, `square(sum(B, log(B)))`, `square(sum(B, log(C)))`, `square(sum(C, log(A)))`, `square(sum(C, log(B)))`, `square(sum(C, log(C)))`, `square(sum(log(A), log(B)))`, `square(sum(log(A), log(C)))`, `square(sum(log(B), log(C)))`	42

Table 3.1: Feature Growth with Composed Operations

Even with such measures to reduce redundancy, the number of numeric features grows as $O(n)$ for `Operations` like `log` and $O(n^2)$ for `Operations` like `sum`. To prevent extreme computational burdens, limit parameters such as `two_arg_feature_limit` are available and configurable. The `Operation` `feature_selection` is also available

to trim generated datasets.

3.3 Learning Algorithm

The algorithm that controls exploration of the `Graph` is a reinforcement learning algorithm mostly taken from [1], originally inspired by [10] (Watkins, Dayan 1992). More specifically, we use Q-learning with linear function approximation, where we approximate the Q values as

$$Q(s,t) = w^t \cdot f(s)$$

Here, w^t, the algorithm's weights of transformation t, controls how strongly each component of $f(s)$ contributes to the estimated value of a state. The vector $f(s)$ characterizing each state for a potential `Node` is a function of the following attributes of the state:

- `Node` modeling performance
- Average reward of transformation in the `Graph`
- Frequency of t in paths to this `Node`
- `Node` accuracy gain from its parents
- `Node` depth
- Budget remaining
- Number of features in this `Node` relative to the root
- Feature types in the dataset
- Whether the transformation is `cherrypick`

Note that there is still some dependence on the transformation t inherent in $f(s)$. The optimal policy simply selects the transformation t which yields the highest Q-value on each iteration based on the current state s.

The weights w^t are derived during training. We start with a set of initial weights for each `Operation` (by default, all 1). After each iteration, we calculate a reward r_i for the algorithm based on the performance of the new `Node` as follows:

$$r_i = \max(0, P_i - \max_{j \in \{0,\ldots,i-1\}} P_j)$$

where P_k is the calculated performance of `Node` k. Thus the algorithm only receives a reward when discovering a `Node` with better performance than all previously explored `Nodes`. This reward is then used to update w_t on iteration i:

$$w_i^t \leftarrow w_{i-1}^t + \alpha \cdot (r_i + \gamma \cdot \phi(\max_{n',t'} Q(s', t')) - Q(s, t)) \cdot f(s)$$

Here α and γ are the learning and discount rates, respectively, configurable as `alpha` and `gamma`. The maximization in the weight update is done over all potential valid (node, transformation) pairs in the next weight update. $\phi(\cdot)$ is a clipping function that bounds the predicted Q in the next iteration to regulate weight updates and ensure they converge; the default implementation of ϕ uses the bounds 0 and $0.1 \cdot (1 - P)$, where P is the current best performance in the `Graph`.

Lastly, to ensure that the algorithm is exposed to enough of the possible state space, the learning algorithm takes a random (valid) action with probability ϵ. Note that this is only done during training; during testing, the policy is to always take the action with the highest expected reward.

Chapter 4

Algorithm Modifications and New Features

On top of the core algorithm, `rl_feature_eng` includes a number of additions that better suit certain use cases, as well as new features that can improve modeling performance. They are described in the following sections.

4.1 Scoring Modification

With the original scoring formulation, there is an imbalance that arises when incorporating improvements in performance into weight updates. As an example, consider datasets `A` and `B`, with base node classification performances of 0.2 and 0.9, respectively. Upon discovering a new set of features that achieves perfect performance, the total reward given to the algorithm over `A` and `B` would then be 0.8 and 0.1, respectively. As a result, the improvements in classification performance over B will have a much smaller impact on the algorithm weights than that of A.

To address this imbalance, we use a scoring system that considers performance relative to the root node to evaluate the performance of each node. First, the raw score R_i of each node is calculated: for classification problems, this is the median Macro F_1 score over 5-fold stratified cross validation, while for regression problems this is the median RMSE over 5-fold cross validation. Then, node performance is

defined as the percentage of possible improvement relative to R_0, the raw score of the root node. Thus, for classification problems, where the optimal performance is an F_1 score of 1, node performance is defined as:

$$P_i = 1 - \frac{1 - R_i}{1 - R_0}$$

For example, given a root raw score of 0.4 and a new raw score of 0.8, the new node performance is $1 - \frac{1-0.8}{1-0.4} = 0.667$, representing 66.7% of the possible improvement.

For regression problems, where the optimal performance is an RMSE of 0, node performance is defined as:

$$P_i = 1 - \frac{R_i}{R_0}$$

So, for a root raw score of 40 and a new raw score of 10, the new node performance is $1 - \frac{10}{40} = 0.75$, representing 75% of the possible improvement.

With these scoring definitions, a performance of 1 represents perfect performance on a dataset, while a performance of 0 always represents no improvement over the root node.

4.2 Automatic Feature Selection

One significant issue with the original feature exploration algorithm is that the number of features in each node is unconstrained. For example, a dataset with 10 numeric features grows to 100 features (10 original $+ 2 * \binom{10}{2} = 90$ new features) after just one `divide` operation. This number quickly grows well into the thousands, or even tens of thousands, when adding more operations, particularly those generating $O(n^2)$ new features like `divide`.

This phenomenon of feature explosion has multiple drawbacks. First, our results become far less explainable if the number of features we generate is an extreme increase over the original dataset. While we can easily interpret the addition of 3

new features to a dataset with 10 features, the importance of 90 additional features is not as clear.

Keeping such a large number of features also greatly reduces computational efficiency. Without a consistent restriction on the number of features in each node, generating new nodes becomes increasingly computationally intensive, given the exponential growth in the number of features when chaining transformations. Furthermore, evaluating the performance of nodes with large number of features is much slower and more memory intensive.

This problem is partially addressed through the inclusion of the `feature_selection` operation, and indeed, in [1] experimental results show that including feature selection as an operation greatly improves performance. However, there is no guarantee that any given node, including the best performing one, will use feature selection.

To address this issue of feature explosion, `rl_feature_eng` includes by default automatic feature selection, set with the `auto_fs` parameter. Automatic feature selection ("auto-fs"), in this context, means requiring feature selection at each node where the number of features exceeds a predetermined cap, determined by the number of features in the original dataset and algorithm settings (specifically, `fs_scaling` and `fs_cap_factor`). The exact method is identical to the operation of the `feature_selection` operation, described in Appendix A.

An unfortunate consequence of using automatic feature selection is that computation time must be spent determining which features to keep. This is not necessarily a net loss efficiency-wise: with automatic feature selection, deeper nodes generally have significantly fewer features, reducing overall computational load. However, the amount of time spent during feature selection can still be rather significant.

We can save computation time by ending model training prematurely during feature selection. As our only goal during feature selection is to reduce the size of the dataset, we have no need to train until the point where model performance converges, as long as we have sufficient confidence that the features we retain are the most important for model performance. Since we use `Random Forest` [5] (L. Breiman,

2001) or `XGBoost` [6] (Chen, Guestrin 2016) for our feature selection method, this corresponds to using fewer decision trees in the model. The number of decision trees we use during automatic feature selection is configurable through the setting `auto_fs_ntrees` (default 10). This parameter can be increased accordingly in order to more confidently identify the most important set of features for model performance, or decreased if maximizing the speed of feature selection is more important.

4.3 Budgeting by Time

The original exploration algorithm expresses exploration budgets in terms of a given number of nodes. Indeed, this behavior can still be enabled in `rl_feature_eng` through the settings `train_budgets` and `test_budget`. However, in practice, we are generally more concerned with quickly finding features in terms of absolute time, rather than a number of iterations of the feature engineering algorithm. Thus by specifying the `training_time_budgets` and `test_time_budgets` settings, the algorithm can be configured to operate based on absolute time. Note that with these options enabled, the feature corresponding to the amount of budget remaining is also based on absolute time.

4.4 The `cherrypick` Operation

In the original algorithm, we already have the `union` operation which can be used to combine promising sets of generated features in the hope of exploiting any synergy between them. However, if we have many promising nodes, some of which perhaps at greater depths, taking the `union` to combine them may either be too slow to generate (by having to chain multiple `union` operations in sequence) or may not be possible given our algorithm configuration (e.g. due to the `max_height` parameter).

To address this desire to combine promising nodes, we introduce the `cherrypick` operation, which operates on between 3 and `cherrypick_size` of the best performing nodes. By default, there are some restrictions on which nodes are eligible for cherrypicking: for example, a high performing node with worse performance than

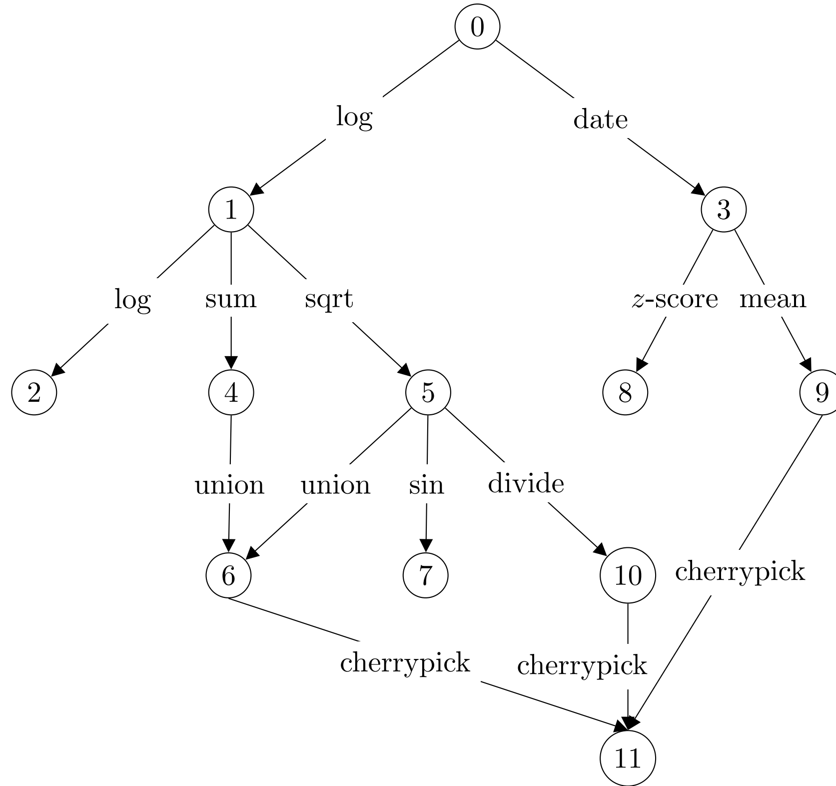

Figure 4.1: Example Graph with 12 Nodes and `cherrypick`

Nodes are labeled in chronological order of addition to the graph.

its even higher performing parent is ineligible. More details on `cherrypick` can be found in Appendix A.

An example `Graph` with 12 `Nodes`, including a `cherrypick` `Operation`, is depicted in Figure 4.1. The numbers within the `Nodes` correspond to the order in which they were added to the `Graph`.

4.5 Runtime Estimation

Naturally, if we configure the budget to be time-based, it makes sense to have the algorithm account for this when selecting operations. In particular, when budgeting by node count, all operations have the same cost: one fewer possible node to explore.

When budgeting by time, this is no longer true. For example, a transformation generating $O(n^2)$ features, like `sum`, will generally require more time both to generate the new dataframe and evaluate its performance than an operation generating $O(n)$ features, like `log`. This implies that selecting the transformation with the highest estimated Q-value may not optimal if it requires too much time relative to other operations.

In order to address this issue, `rl_feature_eng` includes the configurable setting `account_for_modeling_time` which makes the exploration algorithm choose the transformation with the highest Q-to-time ratio at any given step. This leaves the Q-value definitions and updates identical but allows us to choose actions with the highest rate of increasing expected value.

Of course, the effectiveness of this mode of operation is complicated by the fact that estimates for the runtime of various transformations are not readily available. There are four sources of runtime cost during each iteration which comprise the bulk of the time spent during each iteration:

1. Search for the most promising transformation
2. Application of the transformation to create a new dataframe
3. Reduction of the number of features during automatic feature selection, if enabled and necessary
4. Model training and evaluation

The time spent in Step 1 increases when accounting for the runtime of each transformation but is a constant cost regardless of the chosen transformation. The runtime of Step 2 depends greatly on the transformation, as an operation like `sum` which generates $O(n^2)$ features requires much more computation than an $O(n)$ operation like `log`. Feature selection in Step 3 can also require much time if the number of features generated in Step 2 is large, though this can be mitigated by using a smaller value for `auto_fs_ntrees`. Lastly, Step 4, the "modeling time", often is the slowest and takes up the majority of the time to execute.

As Step 4 is generally the slowest, enabling the `account_for_modeling_time` option instructs the algorithm to estimate its runtime by constructing and evaluating the new `Node` for a given operation by using a much smaller number of decision trees during modeling (default 10, configurable through `modeling_time_est_ntrees`). As we must construct a dataframe and partially train a model to do this, this requires a large amount of time when done for each individual valid transformation, so we make some assumptions about expected runtimes and make the following approximations:

- In general, operations within a given class (e.g. `one_arg`) require the same amount of time
- `divide` requires twice as much time as other `two_arg` operations, as it is the only `two_arg` operation lacking symmetry
- `union` and `cherrypick` modeling runtimes scale with the total number of features among the constituent nodes

Thus, with these approximations, we can cache results for each operation type and save time calculating the runtimes for other similar operations.

4.6 Model Selection

Another addition available in `rl_feature_eng` is the option to choose between two different learning algorithms: `Random Forest` and `XGBoost`. Each brings unique advantages, and the user can decide which one to choose. By default, the chosen learning algorithm is used for both evaluating `Node` performance and instances of feature selection.

A big advantage of `Random Forest` is its ease of use and generally strong performance "out-of-the-box", i.e. without hyperparameter tuning. Thus in `rl_feature_eng` all `Random Forest` parameters are kept at default values. This ease of use and straightforwardness also is the reason why `Random Forest` is chosen by default, with the setting `algo_setting = "rf"`.

`XGBoost`, on the other hand, occupies a different niche. It is often blazingly fast, which often means being able to explore many more nodes in the transformation graph than when using `Random Forest`. Using `XGBoost` can be enabled with the setting `algo_setting = "xgb"`. However, it is not as simple to use, as generally, some hyperparameter tuning is required to achieve good performance. This problem further compounds since we are exploring potentially hundreds of nodes on each node, each with a different dataset.

To address this need for hyperparameter tuning, `rl_feature_eng` by default spends a certain proportion of its time budget (`xgb_opt_proportion`, default `0.1`) optimizing hyperparameters for the base dataset. During this optimization, the maximum number of estimators is controlled by `xgb_max_estimators`. Afterwards, it stores the optimal hyperparameter configuration and uses it for the rest of the nodes in the same `Graph`.

Chapter 5

Using `rl_feature_eng`

There are three modes of operation in `rl_feature_eng`: bulk training, bulk testing, and direct testing. We discuss each operation mode in the following sections.

5.1 Training

To ensure that the feature engineering algorithm explores potential transformations efficiently, we must train the algorithm so that the operation weights guide exploration intelligently. Since we aim to have our algorithm perform well over a wide variety of datasets, we train `rl_feature_eng` over multiple datasets during training to improve its generalization capabilities.

To train the algorithm, we first specify a directory of datasets to train over, and then customize, if desired, various training parameters. Some of these parameters control the duration of training (e.g. `train_num_batches`, `train_time_budgets`), while others control the behavior of the learning algorithm (e.g. `alpha`, `epsilon`). The full list of training parameters is available in Appendix B.

During training, the algorithm continually updates weights as it trains on each dataset. The final weights are output to a specified file, and if desired, bulk testing commences with the final weights.

5.2 Bulk Testing

Bulk testing is a similar operation mode to training, except we start with a specified set of weights and maintain them throughout. Furthermore, the policy is to always take the action with the highest Q-value (or Q-to-time ratio if we are using runtime estimation): at no point is a random action taken.

We start, as with training, by specifying a directory of testing datasets and our desired testing parameters. As each dataset is analyzed, the best performing node is shown. If `output_best_test_features` is specified, its dataset is output to file. Once all datasets have been handled, the initial (root node) performance and best node performance are given, along with various statistics describing the testing results.

Note that as we wish to evaluate the performance of our new features on unseen data during testing, a training and testing split must be given for each test dataset. Exploration is done on the training portion of each dataset, and final testing performance is evaluated for the root and best node on the test portion of the dataset.

5.3 Direct Testing

Instead of evaluating feature engineering performance on multiple datasets, we can also find promising new engineered features of a specific dataset, perhaps as one step of a larger machine learning routine. In this setting, we use the direct testing functionality. Below is an example showing how to use this mode.

First, import the relevant modules:

```
from rl_feature_eng import engineer_features, FeatureStore, Primitive
```

The `engineer_features` function is the main routine for exploring new features, while the `FeatureStore` object will store the results of the best features we find. We will use the `Primitive` module to generate a new dataframe with the best features we find.

Next, specify any desired hyperparameters and initialize a `FeatureStore` object:

```
params = {"account_for_modeling_time": True, "fs_scaling": "sqrt",
"test_time_budget": 300}
fs = FeatureStore()
```

Next, run the feature engineering routine with the original dataframe and target, target type (`"categorical"` for classification problems and `"regression"` for regression problems), the instantiated `FeatureStore` object, and desired hyperparameters:

```
engineer_features(data=data, target=target, target_type="regression",
feature_store=fs, hyperparams=params)
```

Running `engineer_features` stores the names of the best engineered features in the `FeatureStore` object. We can obtain these names in a `json` format with the following call:

```
names = feature_store.export_json()
```

With the best feature names known, we can now use the `Primitive` module to generate a dataframe with the engineered features:

```
df = Primitive.transform(names)
```

Now `df` contains all the engineered features we found through `engineer_features`.

To extract n sets of features, set the `n` parameter of `FeatureStore` to the desired value and use the method `export_weighted_feature_sets` in `FeatureStore`. This yields up to n sets of feature names, weighted by their estimated quality, as described in the following section.

5.4 Incorporation into Alpine Meadow

As mentioned in the Related Work section, Alpine Meadow is a powerful standalone auto-ML package that emphasizes quickly returning fast, easily interpretable

machine learning pipelines to users. In this section, we describe the process of incorporating `rl_feature_eng` into Alpine Meadow, as an example of what can be done when combining `rl_feature_eng` into larger frameworks.

Alpine Meadow operates by generating a search space of *logical pipelines*, which constitute machine learning pipelines consisting of various steps. The individual components of each pipeline are called *primitives*. Primitives encapsulate individual aspects of a machine learning pipeline: data processing, data transformation, modeling, etc. Indeed, this is where the `Primitive` module of `rl_feature_eng`, which generates a dataframe given a list of names of features to generate, takes its name.

To combine Alpine Meadow and `rl_feature_eng`, we first allocate a specific budget for `rl_feature_eng` to run. Next we run the `engineer_features` routine we saw in the previous section, storing the best results in a `Feature_Store` object.

After finding which sets of features to use, it suffices to add logical pipelines to the default search space for a given dataset. We do this by duplicating each preexisting logical pipeline in the original search space and adding a `FeatureEngineeringPrimitive` step at the beginning of each duplicate. This primitive, which internally calls the `Primitive` module of `rl_feature_eng`, transforms the original dataset into an augmented dataset with the features we found from `rl_feature_eng`. After transformation, we continue with the sequence of primitives originally within the logical pipeline. If the transformed dataframe improves modeling performance, Alpine Meadow will automatically prioritize creating physical pipelines from logical pipelines including the transformation induced from feature engineering.

To encourage feature diversity among the various pipelines, we store multiple (default 5) promising sets of features in the `FeatureStore` object. To retrieve them, we use the `export_weighted_feature_sets` method of `FeatureStore` like so:

```
weighted_feature_sets = fs.export_weighted_feature_sets()
```

This outputs a list of `(weight, feature_names)` pairs, corresponding to each set of promising new features found during exploration and their weights, which

measure how promising each new dataset is. To determine the weights, we record the score improvement of each new dataset over the root dataset and apply a softmax function. Thus, the weight for dataset i with improvement d_i over the root dataset is

$$w_i = \frac{e^{d_i}}{\sum_{j=1}^{n} e^{d_j}}$$

To avoid crowding the total search space, we add the augmented feature engineering pipelines to the original search space according to probabilities given by the weights of the features they represent. In this way, we end up retaining more pipelines with the most promising features, while still keeping some pipelines with less promising features to ensure feature diversity.

Chapter 6

Experimental Results

In this chapter we discuss experimental results involving `rl_feature_eng`, both independently and combined with Alpine Meadow.

6.1 Experimental Setup and Results

To evaluate the feature engineering routine in `rl_feature_eng`, we conduct an experiment with 3 total phases: training, validation, and testing. Our objective is to train and evaluate multiple `rl_feature_eng` models, select the most promising ones, and then evaluate them alone and in conjunction with the Alpine Meadow auto-ML system.

During the training phase, we take 150 datasets and train 16 different `rl_feature_eng` weight sets, each using different settings. For all training runs, we use $\alpha = 0.05, \gamma = 0.99, \epsilon = 0.15$, with budgets of 60, 150, 300, or 600 seconds, in random order. Parameter settings are otherwise identical except for four: `algo_setting`, `fs_scaling`, `account_for_modeling_time`, and `op_list`. These settings, respectively, control: whether to use `Random Forest` or `XGBoost` as the model; how strongly to filter features during feature selection; whether to account for modeling time when choosing actions; and which `Operations` to consider during exploration. More detailed explanations of these settings can be found in Appendix B. The distribution of each setting among the 16 trials is given in Table 6.1.

Trial	algo_setting	fs-scaling	Use Modeling Time	Use cherrypick
1	rf	sqrt	N	Y
2	rf	sqrt	N	N
3	rf	sqrt	Y	Y
4	rf	sqrt	Y	N
5	rf	linear	N	Y
6	rf	linear	N	N
7	rf	linear	Y	Y
8	rf	linear	Y	N
9	xgb	sqrt	N	Y
10	xgb	sqrt	N	N
11	xgb	sqrt	Y	Y
12	xgb	sqrt	Y	N
13	xgb	linear	N	Y
14	xgb	linear	N	N
15	xgb	linear	Y	Y
16	xgb	linear	Y	N

Table 6.1: Trial Settings

We use an additional 120 datasets for the validation and testing phases, of which 46 are used during validation. During validation, we compare the performance of our 16 trained weight sets, each with their specific settings, along with basic `Random Forest` and `XGBoost` models.

Given the results of the 18 total methods over the validation datasets, we select two of them to evaluate over the test datasets. To determine which two to select, we calculate the z-scores of performances of each method over each dataset and take the two models with the highest z-score total. This helps normalize performance differences across datasets and ensures we choose the trials with consistently good performance. These results are shown in Table 6.2.

Trial	Average z-score	Average z-score Rank
Base rf	-0.486738501	17
Base xgb	-0.520263653	18
1	0.054513634	8
2	0.04584359	9
3	-0.064747978	13
4	0.109357191	6
5	**0.239124863**	**2**
6	0.175730037	4
7	-0.102349857	14
8	0.163106137	5
9	-0.107983756	15
10	0.003813241	11
11	0.043646142	10
12	-0.10978345	16
13	0.058990208	7
14	-0.020971936	12
15	**0.29237381**	**1**
16	0.226340278	3

Table 6.2: Validation Results

Average z-score is over z-scores calculated over trials on each dataset.

For ranks, 1 is best and 18 is worst. The best two values are bolded.

Table 6.2 shows that the overall best performing models are Trials 5 and 15. Note that the two trials simply using `Random Forest` or `XGBoost` without any feature engineering perform significantly worse than the trials with feature engineering included.

Now we enter the testing phase, where we evaluate and compare 9 different setups involving `rl_feature_eng` and Alpine Meadow. For each setup, we run `rl_feature_eng` for the specified feature engineering budget and pass the discov-

ered features to Alpine Meadow. We then run Alpine Meadow for its corresponding budget and observe the score of the best pipeline found by Alpine Meadow. For the setups where Alpine Meadow is not run, we observe the test split score of the best node found by `rl_feature_eng`. The parameters for each run are the same as in training, respecting the specific settings of models 5 and 15.

To evaluate each setup relative to Alpine Meadow, we make use of a multiple hypothesis testing procedure involving the Bonferroni-Dunn test [7] (Dunn, 1961), as described in [8] (Demar, 2006). We first assign each setup i a rank r_i^j on each dataset j, where rank 1 is best. Tied performances are assigned the average of the ranks they would otherwise be assigned. We then compute R_i, the average of the ranks for an individual setup:

$$R_i = \frac{1}{n} \sum_j r_i^j$$

Here $n = 74$, as there are 74 testing datasets in total.

Next, we calculate the test statistic z_i for each testing setup, relative to Setup A, which only uses Alpine Meadow:

$$z_i = \frac{(R_1 - R_i)}{\sqrt{\frac{k(k+1)}{6n}}}$$

where $k = 9$ is the number of setups. The Bonferroni-Dunn test we employ here is then to compare the p-value induced by test statistic, assuming the normal distribution, to the threshold induced by taking $\alpha = \frac{0.05}{k-1} = 0.00625$.

The budget settings, average z-score, average rank, and test statistic on the testing datasets for each setup are shown in Table 6.3. We see that setup B has the best average z-score among the 9 setups, while setup F has the best average rank. Setups F and B exhibit better rank performance than Setup A, which only uses Alpine Meadow. Notably, setups B and F are the only two that do not incorporate Alpine Meadow whatsoever. However, neither of these setups outperform Alpine Meadow by a significant margin; the margin is close enough to resemble statistical noise, considering that the necessary z_i to produce a significant result is roughly 3.2,

Setup	AM Budget	FE Budget	FE Model	Avg z-score	Avg Rank	z_i
A	10	0	-	0.1653	4.5878	-
B	0	10	5	**0.1962**	4.5810	0.0150
C	5	5	5	-0.1281	5.2770	-1.5307
D	7.5	2.5	5	-0.1240	5.2162	-1.3956
E	9	1	5	0.0431	5.2094	-1.3806
F	0	10	15	0.1578	**4.5405**	**0.1050**
G	5	5	15	-0.0837	5.2094	-1.3806
H	7.5	2.5	15	-0.1021	5.1216	-1.1855
I	9	1	15	-0.1244	5.2567	-1.4857

Table 6.3: Testing Results

Average z-score and average rank are calculated over setups for each test dataset.

For ranks, 1 is best and 9 is worst. Best values are bolded.

Budget values are in minutes.

for $\alpha = 0.00625$. Hence, no alternative setup has achieved a statistically significant result superior to Alpine Meadow.

We give the performance of each testing setup over all 74 testing datasets in Figure 6.1.

6.2 Analysis of Results

While not constituting a formal statistical test, the comparison of bare `Random Forest` and `XGBoost` models to the results of `rl_feature_eng` setups during the validation phase imply that feature engineering does help on many datasets. Of course, we cannot expect every dataset to respond significantly to feature engineering efforts without explicit domain knowledge, so the observation that feature engineering outperformed the base models overall in every validation trial supports the hypothesis that `rl_feature_eng` finds useful features in some proportion of datasets that are responsive to novel features.

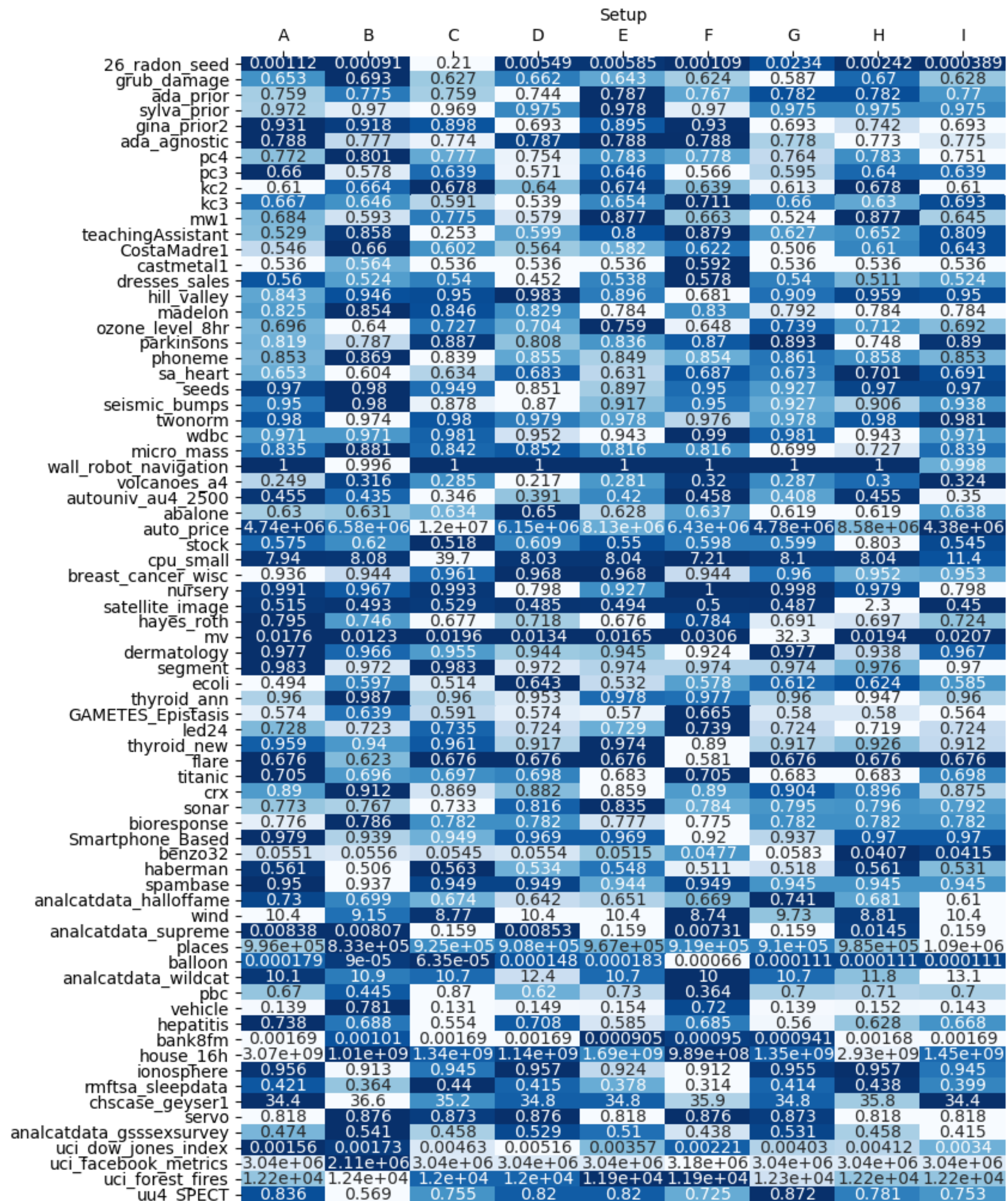

Figure 6.1: Test score heatmap

Each row, corresponding to a different dataset, has its own heatmap color scheme.

Darker colors indicate better performance on a dataset.

It is ultimately not clear that combining `rl_feature_eng` with Alpine Meadow improves modeling performance overall. Interestingly, the experimental setups that only used one method performed better than any setup that combined the two. It seems that setups using only feature engineering or only Alpine Meadow yield comparable performance, with the differences in rankings between these setups over the testing datasets being quite small.

One possible explanation for this gap in performance between the combined and individual setups is a decline in search efficiency when combining the two methods. Alpine Meadow operates by successively improving upon previous pipelines, and using too much of the allotted time for feature engineering could hinder this optimization process. Furthermore, if too many features are retained during feature engineering, fewer pipelines can be explored due to an increase in computational load. This could possibly be mitigated by a more aggressive feature selection routine or a more sophisticated method of integrating Alpine Meadow and `rl_feature_eng`.

Chapter 7

Extensions and Future Work

As automated feature engineering is a burgeoning young field, there is an enormous number of potential directions to explore as for improving upon current methods. Here, we discuss what improvements could be made specifically to `rl_feature_eng` to improve the utility it has in feature engineering tasks.

Of course, there are potentially significant improvements to be made regarding parameter tuning. This includes tuning learning parameters such as `alpha` or `epsilon` to improve exploration, and feature filtering parameters like `two_arg_corr_threshold` and `fs_scaling` to reduce the number of redundant and uninformative features which occupy computational resources for little benefit.

More interesting than parameter tuning are new ideas that could greatly improve search efficiency. One is the idea of search pruning, where we gradually restrict the search space over possible transformations over time. This becomes important when running `rl_feature_eng` for long periods of time, as the entire search space must be recalculated on each iteration (due to e.g. changing proportion of budget remaining), which can have a significant negative effect on algorithm runtime in later stages. Pruning transformations that are unlikely to ever be selected could therefore reduce the amount of time we spend searching and evaluating transformations that are unlikely to be chosen.

Another possible efficiency improvement relies on exploiting the `warm_start` pa-

rameter of models like `Random Forest`. As we are fitting a large quantity of models during operation, being able to terminate model evaluation early in certain situations (e.g. if the node is almost certainly low-performing) could greatly increase the number of nodes we explore in a given time frame. This would require some investigation into modeling the behavior of `Random Forest` and `XGBoost` performance over the number of estimators present; the work in [9] (Probst, Boulesteix 2017) regarding tuning random forest hyperparameters may be useful in this regard.

Yet another efficiency improvement would be the exploitation of multiple cores in the exploration algorithm. In this work, we only use one core for all experimental runs with `rl_feature_eng`, but configuring the algorithm to manage multiple worker units, each exploring separate parts of the tree, should yield a great improvement in search capability.

Lastly, a large potential improvement could be had from incorporating `rl_feature_eng` into Alpine Meadow in a more sophisticated manner. The current incorporation method simply runs `rl_feature_eng` and Alpine Meadow sequentially. A dynamic budget allocation, where feature engineering is stopped upon reaching a termination condition (e.g. once node performance stalls) could ensure that the budget is being spent in the most efficient way possible. An intermediary feature selection routine, ensuring that Alpine Meadow only operates upon demonstrably useful features, could also help improve overall performance.

Chapter 8

Conclusion

In this work we present `rl_feature_eng`, a Python package capable of automatically engineering promising features for a given dataset. We describe the works inspiring it, the motivations for improvements and additions we implement to make it suitable for use in larger auto-ML frameworks, and its structure, function, and customization opportunities.

While ultimately the experiments regarding incorporation of `rl_feature_eng` into Alpine Meadow do not yield any statistically significant improvements, the results seen in this work support the idea that `rl_feature_eng` is a promising tool that can aid in the analysis of datasets where feature engineering is a promising avenue of improvement. Many future opportunities for improvements, including some already mentioned in this text, could make `rl_feature_eng` suitable for eventual inclusion in larger auto-ML frameworks.

Bibliography

[1] Khurana, U., Samulowitz, H., and Turaga, D. (2018, April). Feature engineering for predictive modeling using reinforcement learning. In Thirty-Second AAAI Conference on Artificial Intelligence.

[2] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... and Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. the Journal of Machine Learning Research, 12, 2825-2830.

[3] Shang, Z., Zgraggen, E., Buratti, B., Kossmann, F., Eichmann, P., Chung, Y., ... and Kraska, T. (2019, June). Democratizing data science through interactive curation of ml pipelines. In Proceedings of the 2019 International Conference on Management of Data (pp. 1171-1188).

[4] Kanter, J. M., and Veeramachaneni, K. (2015, October). Deep feature synthesis: Towards automating data science endeavors. In 2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA) (pp. 1-10). IEEE.

[5] Breiman, L. (2001). Random forests. Machine learning, 45(1), 5-32.

[6] Chen, T., and Guestrin, C. (2016, August). Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining (pp. 785-794).

[7] Dunn, O. J. (1961). Multiple comparisons among means. Journal of the American statistical association, 56(293), 52-64.

[8] Demar, J. (2006). Statistical comparisons of classifiers over multiple data sets. Journal of Machine learning research, 7(Jan), 1-30.

[9] Probst, P., and Boulesteix, A. L. (2017). To tune or not to tune the number of trees in random forest. The Journal of Machine Learning Research, 18(1), 6673-6690.

[10] Watkins, Christopher JCH, and Peter Dayan. “Q-learning.” Machine learning 8.3-4 (1992): 279-292.

Appendix A

Operation Types

In `rl_feature_eng`, each `Operation` is classified by type. Below the various operation types are explained, along with the members of each operation class. Most of the operations are taken from [1], though not all operation definitions between `rl_feature_eng` and [1] are identical. `cherrypick` is a new operation type.

- `one_arg`: Operations that act upon a single numeric feature element-wise, i.e. functions of the form $f : (c_1, \ldots, c_n) \to (f(c_1), \ldots, f(c_n))$.

  This class includes log, sin, cos, tanh, reciprocal (rc), sigmoid, square, and square root (sqrt).

- `two_arg`: Operations that act upon two numeric features element-wise, i.e. functions of the form $f : (c_1, \ldots, c_n) \times (d_1, \ldots, d_n) \to (f(c_1, d_1), \ldots, f(c_n, d_n))$.

  This class includes sum, subtract, multiply, and divide. To reduce redundancy, the operations sum, subtract and multiply are treated as symmetric, and only one of $f(c_i, d_i)$ and $f(d_i, c_i)$ is generated (while subtraction is not mathematically symmetric, there is no benefit to retaining both f and $-f$ as features.)

- `statistical`: Operations that act upon a single numeric feature as a whole and output statistics based on the feature, i.e. functions of the form
  $f : (c_1, \ldots, c_n) \to (f_1(c_1, \ldots, c_n), \ldots, f_n(c_1, \ldots, c_n))$.

This class includes z_score, standard deviation, min_max_norm (normalizing a single feature to between 0 and 1), binning_u (uniform binning with 10 bins), and binning_d (dynamic binning dependent on data).

- `aggregate`: Operations that act upon a numeric feature class-wise according to the classes in a categorical feature.

  As an example, given a numeric feature `pollution_index` and a categorical feature `day_of_week`, the `max` aggregate operation will generate one feature where the `pollution_index` value for each day is replaced by the maximum `pollution_index` value on that day in the entire dataset. Other examples would include computing a statistic for each client of a business or each country in a dataset.

  This class includes max, min, count, standard deviation (std), z_score, and mean. All aggregate operation names are suffixed by `_agg` in `rl_feature_eng`.

- `frequency`: There is only one operation in this class, `one_term_frequency`. This operation takes a numeric feature and outputs a feature corresponding to the frequency of each value as a proportion of the number of instances.

- `date_split`: Parses a `np.datetime` feature and generates several features based on the contained dates. These are : `["year", "month", "day", "hour", "minute", "second", "microsecond", "nanosecond", "dayofweek"]`

- `union`: A `union` operation simply takes two dataframes and produces the mathematical union of their features.

  To avoid redundancy, certain pairs of nodes are forbidden from being joined through a `union` operation, e.g. a node and its parent.

- `compact_one_hot`: An operation that attempts to join low frequency one-hot categories for categorical variables into a “misc” category. Not used by default.

- `feature_selection`: Performs feature selection on a given node. While there are many methods for performing feature selection, the feature selection algorithm implemented here uses the `feature_importances_` attribute of either a

`RandomForest` or `XGBoost` classifier, depending on the `algo_setting` parameter, to choose which features to retain.

Once the feature selection model is chosen, we then perform feature selection by selecting $N = c \cdot \phi(n)$ features, where c is the `fs_cap_factor` setting, $\phi(\cdot)$ is given by the `fs_scaling` setting, and n is the number of features in the original dataset. Thus, for $c = 1, \phi(\cdot) = \sqrt{\cdot}$ and $n = 100$, 10 total features will be kept. Afterwards, all original features are re-added to the node dataframe. Thus, after feature selection, we will have retained all the original features and kept anywhere between 0 and 10 engineered features, depending on how well they ranked in terms of feature importance in our selected model.

- `cherrypick`: An operation, similar to `union`, which joins features from up to `cherrypick_size` nodes. Unlike `union`, `cherrypick` has several unique constraints:
  - A minimum of 3 nodes must be joined.
  - Only nodes with better performance than all of their ancestors can be joined (i.e. "good" nodes)
  - Only the qualifying nodes with the absolute best performance can be chosen

  These constraints imply that `cherrypick` can only be called once after finding three high-performing nodes, and then only once each time after finding another qualifying high-performing node.

Appendix B

Settings and Parameters

rl_feature_eng has a wide range of settings which can be tweaked by the user to suit their individual needs. We categorize them into multiple types of settings, given below.

B.1 Operational Settings

These settings affect the setup and output behavior for rl_feature_eng. This includes reading in parameters and outputting certain information.

- config_settings_file [str]: File name containing a json dictionary mapping settings to values. If specified, will output a json file in the same directory containing all settings.
- skip_training [bool]: Do not do training (if specified, must input training weights)
- train_weights_input_file_name [str]: File containing pickled training weights for each operation
- output_training_weights [bool]: Output training weights after training
- training_output_name [str]: Name for output training weights

- skip_testing [bool]: Do not do testing (i.e. only get training weights)
- output_best_test_features [bool]: Output the dataframes corresponding to the best sets of features found during testing
- test_weights_file_name [str]: Name of file containing best features found during testing
- performance_evolution_name [str]: Name of file containing the index of the best performing node over time

B.2 Algorithm Settings

The following settings alter the modeling and exploration behavior of rl_feature_eng.

- op_list [list[str]]: specify which types of operations to include (Options include ["one_arg", "two_arg", "statistical", "aggregate", "frequency", "compact_one_hot", "feature_selection", "cherrypick", "union", "date_split"])
- algo_setting [str] (One of ["rf", "xgb", "rf_multiclass"]): Learning algorithm to use, between Random Forest and XGBoost. "rf_multiclass" uses Random Forest for multiclassification problems and XGBoost for regression and binary classification.
- account_for_modeling_time [bool]: Estimate and account for modeling time when selecting new transformations
- xgb_opt_proportion [float]: Proportion of time we optimize hyperparameters when using xgboost
- xgb_max_estimators [int]: Max number of estimators used in xgboost modeling
- train_budgets [list[int]]: Maximum number of nodes to explore for each training run

- train_time_budgets [list[int]]: Time budget for each training run
- train_num_batches [int]: Number of times to repeat each training run
- alpha [float]: Learning rate during training
- gamma [float]: Discount factor during training
- epsilon [float]: Frequency of random actions during training
- test_budget [int]: Maximum number of nodes to explore during testing
- test_time_budget [int]: Time budget for testing
- modeling_time_ntrees [int]: How many trees to use for Random Forest when estimating modeling time
- auto_fs_ntrees [int]: How many trees to use for Random Forest when doing automatic feature selection
- preprocessing_opt_outs [list[str]]: Which preprocessing steps to skip

  (Options: "skip_all", "skip_drop_index", "skip_infer_dates", "skip_remove_full_NA_columns", "skip_fill_in_categorical_NAs", "skip_impute_with_median", "skip_one_hot_encode", "skip_remove_high_cardinality_cat_vars", "skip_rename_for_xgb")
- max_height [int]: Maximum height of the exploration graph

  (Note: cherrypick and feature_selection are not constrained by this limit)
- two_arg_feature_limit [int]: Maximum number of features before two_arg operations are forbidden due to computational constraints
- agg_product_feature_limit [int]: Maximum product of number of categorical and numerical features before aggregate operations are forbidden
- node_total_data_limit [int]: Maximum product of number of instances and features before feature generating operations are forbidden

- `fs_cap_factor [int]`: What scaling factor to use for limiting the number of retained features during auto-fs

- `fs_scaling [str]`: Scaling used to determine max number of features returned during automatic feature selection

- `two_arg_corr_threshold [float]`: Threshold for removing correlated operations

- `auto_fs [bool]`: Whether to do automatic feature selection

- `cherrypick_size [int]`: Maximum number of nodes to use for `cherrypick` operations.

B.3 Performance Settings

Below are various settings that can improve the performance of `rl_feature_eng` under specific circumstances.

- `calculate_from_scratch [bool]`: If `True`, discard the dataframes for each explored `Node` after evaluation. When constructing a new `Node`, all parent dataframes must be recalculated. This incurs a cost in runtime while minimizing memory usage.

- `keep_feature_cache [bool]`: Whether to maintain a cache of generated features, so as to avoid redundant calculations when using the same operation on a different node. Useful for saving time when memory is plentiful.

- `n_cores [int]`: Number of cores to use. This is passed to the relevant underlying models (i.e. sklearn `RandomForest` and `xgboost`). -1 indicates to use all available cores.

B.4 Output Settings

- `show_warnings [bool]`: Show warnings (e.g. from scikit-learn) during operation.
- `verbose [int]`: Show output, with higher integer values yielding increasingly more output. 0 means silent.

Appendix C

Package Structure

In this section, we discuss the general structure of `rl_feature_eng`. This can be useful for those wanting to extend the package, e.g. by implementing their own operations or altering the exploration behavior.

The base module of `rl_feature_eng` is the `Main` module. Indeed, the `engineer_features` function called in the previous section is defined in `Main`. `Main` reads in command line arguments and determines, based on our method of calling it, whether to use data passed in as arguments or data residing in specific folders on the file system. `Main` also handles initializing the `Config` object and setting some of its parameters appropriately based on the functional mode.

Once `Main` has processed the inputs appropriately, it passes control to `PipelineManager`, which handles various operational aspects of the chosen functional mode(s). This includes tasks like reading in initial training weights and calling the appropriate training and testing algorithms.

The actual training and testing algorithms for exploring new features reside in `RL1`, which takes its name from the corresponding algorithm in [1]. `RL1` handles tasks such as ensuring the algorithm doesn't overspend its budget, updating the `Graph` object corresponding to the exploration graph, storing high-performing engineered features in a given `FeatureStore` object, and maintaining and updating operation weights during training. The features corresponding to each weight are defined in

the `Characteristics` module.

A `Graph` object maintains a group of `Node` objects, with methods to perform functions like adding certain types of nodes and validating whether an action would produce a valid `Node`. Each `Node` maintains information related to its children and parents, while also maintaining a `FeatureSet` object which stores the information related to the task: the dataframe corresponding to this `Node`, the target, division into training and test sets, etc.

`Transformer` handles the administration of generating new features, which calls the appropriate `Operation` needed to transform a dataframe. The `operations` module contains each `Operation`, each of which implement a `transform` method to act upon a dataframe. The mathematical operations underpinning each `Operation` are defined in the `TransformOperations` module.

Lastly, given a list of names of features to engineer (e.g. from a `FeatureStore` object), the `transform` method of the `Primitive` module can add them to a given dataframe. In general the feature names, which can be nested, must conform to the format `operation(arg1, arg2, ...)`, e.g. `log(temperature)` or `divide(weight, square(height))`. For `DateSplitOperation`, the feature names have the format `date_split_year(date)`, `date_split_month(date)`, etc.

Appendix D

Preprocessing

By default, in rl_feature_eng various preprocessing steps are taken for each dataset. All of them can be opted out of if desired, though care should be taken to ensure that data is appropriately cleaned beforehand when skipping certain steps. Preprocessing steps to skip can be specified through the preprocessing_opt_outs setting, which can include any of the values given below. The definitions for each option (except for skip_all) give the default behavior which can be turned off through specifying the corresponding option.

- skip_all: Skip all preprocessing steps.
- skip_drop_index: If using a d3m dataset, remove the d3mIndex feature.
- skip_infer_dates: By default, rl_feature_eng tries to infer a datetime feature from any feature whose name includes the string "date". Skipping this step can prevent rl_feature_eng from mistakenly inferring a datetime feature.
- skip_remove_full_NA_columns: Remove any column which only contains NA values.
- skip_fill_in_categorical_NAs: Replace NA values in categorical columns with the value __missing__.

- `skip_impute_with_median`: Replace `NA` values in numerical columns with median imputation.

- `skip_one_hot_encode`: One hot encode categorical features.

- `skip_remove_high_cardinality_cat_vars`: Remove categorical variables with too high a ratio of categories relative to total instance count (default 0.8). This avoids situations where each value is unique and any generated features based on the column categories will be uninformative.

- `skip_rename_for_xgb`: Don't rename features which contain illegal characters when using `XGBoost`. All name changes are reverted after finishing operation, but `XGBoost` throws errors during operation if a feature name contains certain illegal characters.

Appendix E

Feature Filtering

By default, certain kinds of features are removed or omitted during operation of `rl_feature_eng`. These features are generally redundant or are unlikely to be informative. The various omissions are described in the following sections.

E.1 Inverse Pairs

Some kinds of operations are inverses of each other, and applying them both on a feature doesn't yield any information. For example, if `X` is a feature, then `square(sqrt(X))` is not an informative feature. Such features are automatically omitted.

E.2 Highly Correlated Features

`two_arg` operations can often generate highly correlated features that don't add much value to performance and distort feature importance rankings, while also requiring unnecessary computational effort. For example, consider feature $A \approx 100$ and feature $B \approx 1$; `sum(A, B)` will be extremely highly correlated to A and is unlikely to be useful. The `two_arg_corr_threshold` (default: 0.99) controls how strong this correlation must be, positive or negative, before such features are omitted.

E.3 Redundant Operations

Certain operations are unlikely to be useful when repeated. For example, `date_split` and `zscore` operations don't make any sense when composed with themselves. For such operations, the `is_redundant_to_self` method returns `True` and thus the algorithm avoids composing them directly.

E.4 Constant Features

Some operations sometimes yield features that are constant, depending on the input features. For example, a binary (0 or 1) feature `X` will yield a constant feature `log(X)` as in `rl_feature_eng`, `log(0)` is defined to be 0. Similarly, given a feature `X` with only large positive values, the feature `tanh(X)` will always yield 1. Such features are always omitted after dataset generation.

E.5 High Cardinality Categorical Features

As `rl_feature_eng` does not currently contain any features operating specifically on categorical features, the only relevant attribute of categorical features are the partitions they induce through their values for `aggregate` oprations. Thus, a categorical feature with very small partitions is unlikely to be useful for generating informative features. For example, consider an "ID" feature for hospital patients; if each patient has a unique ID, then one hot encoding the ID feature will only serve to generate a large number of useless features. Furthermore, this creates redundant computational load. Thus, by default, categorical features with too few unique values are removed by default, though they are restored after operation is complete.