Chapter 9

Automating Feature Engineering in Supervised Learning

Udayan Khurana

IBM Research

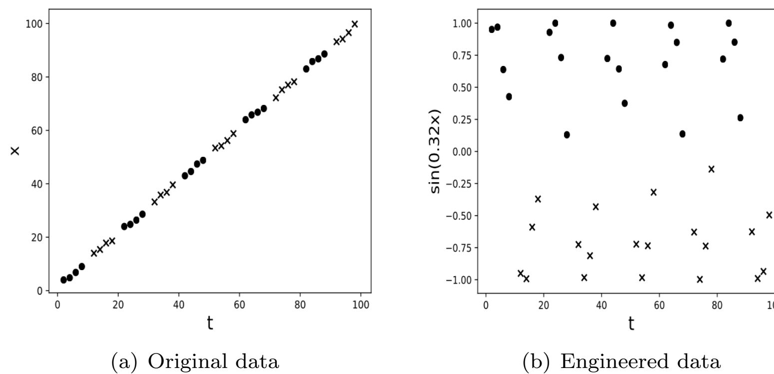
9.1 Introduction 116
 9.1.1 Challenges in Performing Feature Engineering 117
9.2 Terminology and Problem Definition 119
9.3 A Few Simple Approaches 120
9.4 Hierarchical Exploration of Feature Transformations 121
 9.4.1 Transformation Graph 122
 9.4.2 Transformation Graph Exploration 123
9.5 Learning Optimal Traversal Policy 125
 9.5.1 Feature Exploration through Reinforcement Learning 127
9.6 Finding Effective Features without Model Training 129
 9.6.1 Learning to Predict Useful Transformations 131
9.7 Miscellenious 133
 9.7.1 Other Related Work 133
 9.7.2 Research Opportunities 134
 9.7.3 Resources 134

Abstract

The process of predictive modeling requires extensive feature engineering. It often involves the transformation of given feature space, typically using mathematical functions, with the objective of reducing the modeling error for a given target. However, there is no well-defined basis for performing effective feature engineering. It involves domain knowledge, intuition, and most of all, a lengthy process of trial and error. The human attention involved in overseeing this process significantly influences the cost of model generation. Moreover, when the data presented is not well described and labeled, effective manual feature engineering becomes an even more prohibitive task. In this chapter, we discuss ways to algorithmically tackle the problem of feature engineering using transformation functions in the context of supervised learning.

9.1 Introduction

Feature representation plays an important role in the effectiveness of a supervised learning algorithm. For instance, Figure 9.1 depicts two different representations for points belonging to a binary classification dataset. On the left, the instances corresponding to the two classes appear to be present in alternating small clusters along a straight line. For most machine learning algorithms, it is hard to draw a classifier separating the two classes on this representation. However, if the feature x is replaced by its *sine*, as seen in the image on the right, it makes the two classes easily separable. Feature engineering is that *task* or *process* of altering the feature representation of a predictive modeling problem, in order to better fit a training algorithm. The *sine* function is a *transformation* function used to perform feature engineering.



(a) Original data (b) Engineered data

FIGURE 9.1: Illustration of two representations of a feature.

Consider the problem of modeling the heart diseases of patients based upon their characteristics such as height, weight, waist, hip, age, gender, amongst others. While the given features serve as important signals to classify the risk of a person, more effective measures, such as BMI (body mass index), and a waist to hip ratio, are actually functions of these base features. To derive BMI, two transformation functions are used – *division* and *square*. Composing new features using multiple functions and from multiple base features is quite common. Consider another example of predicting hourly biking rental count [1] in Figure 9.2. The given features lead to a weak prediction model. However, the addition of several derived features dramatically decreases modeling error. The new features are derived using well known mathematical functions such as *log*, *reciprocal*, and statistical transformations such as *zscore*. Of-

[1] Kaggle bike sharing: https://www.kaggle.com/c/bike-sharing-demand

datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	count
2011-01-01 01:00:00	1	0	0	1	9.84	14.395	81	0	16
2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0	40
2011-01-01 03:00:00	1	0	0	1	9.02	13.635	80	1	32

(a) Original features and target (count).

year(datetime)	hour(datetime)	log(humidity)	log(windspeed)	1/humidity	1/log(hour(datetime))	zscore(datetime)
2011	1	4.394449	0	0.0125	0	-1.711228
2011	2	4.382027	0	0.0123	3.3219	-1.711030
2011	3	4.382027	0	0.0123	2.0959	-1.710832

(b) Additionally engineered features using transformation functions.

FIGURE 9.2: In Kaggle's biking rental count prediction dataset using Random Forest regressor, the addition of new features reduced the Relative Absolute Error from 0.61 to 0.20.

ten, less known domain-specific functions prove to be particularly useful in deriving meaningful features as well. For instance, *spatial aggregation*, *temporal windowing*, are heavily used in spatial and temporal data, respectively. A combination of those – *spatio-temporal aggregation*, can be seen in the problem of predicting rainfall quantities from atmospheric data. The use of the recent weather observations at a station, as well as surrounding stations greatly enhance the quality of a model for predicting precipitation. Such features might not be directly available and need aggregation from within the same dataset [2]

Feature engineering may be viewed as the addition or removal of features to a dataset in order to reduce the modeling error. The removal of a subset of features, called *dimensionality reduction* or *feature selection* is a relatively well studied problem in machine learning [7] [16]. The techniques presented in this chapter focus on the feature construction aspects while utilizing feature selection as a black-box. In this chapter, we talk about general frameworks to automatically perform feature engineering in supervised learning through a set of transformation functions. The algorithms used in the frameworks are independent of the actual transformations being applied, and are hence domain-independent. We being with somewhat simple approaches for automation, moving on to complex performance-driven, trial and error style algorithms. We then talk about optimizing such an algorithm using reinforcement learning, concluding with an approach that learns patterns between feature distributions and effective transformations. First of all, let us talk about what makes either manual or automated feature engineering challenging.

[2]NOAA climate datasets: https://www.ncdc.noaa.gov/cdo-web/datasets

9.1.1 Challenges in Performing Feature Engineering

In practice, feature engineering is orchestrated by a data scientist, using hunch, intuition and domain knowledge. Simultaneously, it involves continuous observation and reaction to the evolution of model performance, in a manner of trial and error. For instance, upon glancing at the biking rental prediction dataset described previously, a data scientist might think of discovering seasonal or daily (day of the week) or hourly patterns. Such insights are obtained by virtue of some past knowledge, obtained either through personal experience or an academic expertise. It is natural for humans to argue that the demand for bike rental has a correlation to the work schedules of people, as well as some relationship to the weather, and so on. This is a collective example of the data scientist applying *hunch*, *intuition*, and *domain expertise*. Now, all of the proposed patterns do not end up being true or useful in model building. The person conducting the model building exercise would actually try the different options (either independently, or in a certain combinations) by adding new features obtained through transformation functions, followed by training and evaluation. Based on which model trials provide the best performance, the data scientist would deem the corresponding new features useful, and vice-versa. This process is an example of *trial and error*. As a result of this process, feature engineering for supervised learning is often time-consuming, and is also prone to bias and error. Due to this inherent dependence on human decision making, it is colloquially referred to as "*an art/science*" [3] [4], making it non-trivial to automate. Figure 9.4 illustrates an abstract feature engineering process centered around a data scientist.

The automation of FE is challenging computationally, as well as in terms of decision-making. First, the number of possible features that can be constructed is unbounded; the transformations can be composed and applied recursively to features generated by previous transformations. In order to confirm whether a new feature provides value, it requires training and validation of a new model upon including the feature. It is an expensive step and infeasible to perform with respect to each newly constructed feature. In the examples discussed previously, we witnessed the diversity of functions and possible composition of functions to yield the most useful features. The immense plurality of options available makes it infeasible in practice to try out all options computationally. Consider a scenario with merely $t = 10$ transformation functions and $f = 10$ base features; if the transforms are allowed to be applied up to a depth, $d = 5$, the total number of options are, $f \times t^{d+1}$, which is greater than a million choices. If these choices were all evaluated through training and testing, it would take infeasibly large amount of time even for a relatively small dataset. Secondly, feature engineering involves complex decision making, that

[3] http://www.datasciencecentral.com/profiles/blogs/feature-engineering-tips-for-data-scientists

[4] https://codesachin.wordpress.com/2016/06/25/non-mathematical-feature-engineering-techniques-for-data-science/

is based on a variety of factors. Some examples are, prioritization of transformations based on the performance with the given dataset or even based on past experience; or, whether to *explore* different transformations or *exploit* the combinations of the ones that have shown promise thus far on this dataset, and so on. It is non-trivial to articulate the notions or set of rules that are the basis of such decisions. Hence, it is also non-trivial to write programs to perform the same task.

In this chapter, we take a closer look at the automation of the tasks described above for feature engineering in supervised learning using transformation functions. We specifically look at the strategies that automate the trial and error methodology, and those that try to learn patterns of association between features and effective transforms from past experience.

9.2 Terminology and Problem Definition

We are given a predictive modeling task consisting of (1) a set of feature vectors, $F = \{f_1, f_2 \dots f_m\}$; (2) a target vector, y. The nature of y – categorical or continuous, describes whether it is a classification or regression problem, respectively. Considering a suitable learning algorithm L, that is applicable in the context of given y, and a measure of performance, m. We use $A_L^m(F, y)$ to signify the performance of a the model constructed on given data with using the algorithm L through the performance measure m. An example of L is logistic regression for classification and an example of m is average F1-score.

Now consider a set of k transformation functions at our disposal, $\mathcal{T} = \{t_1, t_2 \dots t_k\}$. The application of a unary transformation, t_i, can be represented as, $f_{out} = t_i(f_{in})$, where $f_{in}, f_{out} \in \mathbb{R}^n$, are features of the same dimension. Similar notation extends to binary and k-ary transformations. A variation of the transformations is written with capital letters, such as $T_1, T_2 \dots T_k$. These are applied on a set of features, F, instead of individual features. They symbolize a separate application of the corresponding function t on each input $f \subseteq F$, such that $t(f)$ is a legal and valid feature. Also, for $F_o = T(F_i)$, F_o includes all the newly generated features besides the original features from F_i. For instance, a *Log* transformation applied to a set of ten numerical features, F, will produce ten new output features, $f_o = log(f_i), \forall f_i \in F$. This extends to k-ary functions, which work on k input features. The entire (open) set of features derived directly or recursively from from F using $\mathcal{T}$ is denoted by $\hat{F_{\mathcal{T}}}$.

A '+' operator on two feature sets (associated with the same target y) is a union of the two feature sets, $F_o = F_1 + F_2 = F_1 \cup F_2$, preserving row order. Note that all operations specified on a feature set , $T(F)$, can exchangeably be written for a corresponding dataset, $D = \langle F, y \rangle$, as, $T(D)$, where it is implied that the operation is applied on the corresponding feature set. Also, for a binary, such as sum, $D_o = D_1 + D_2$, it is implied that the target is

common across the operands and the result. Transformations on feature sets add features or keep the set unchanged; on the other hand, a *feature selection* operator removes features. However, it can be written in the same algebraic notation as set transformations, such as $T_2(FS_1(T_1(D_0)))$ or $T_2.FS_1.T_1(D_0)$.

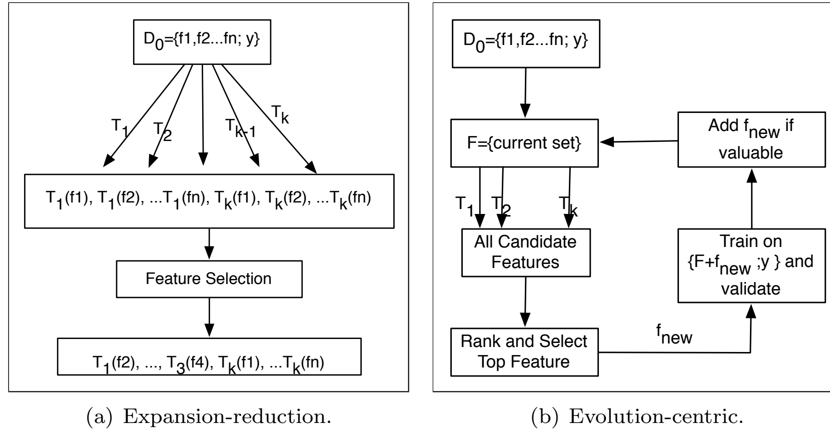
The goal of feature engineering is stated as follows. Given a set of features, F, and target, y, and a set of transformations, $\mathcal{T}$ – find a set of features, $F^* = F_1 \cup F_2$, where $F_1 \subseteq F$ (original) and $F_2 \subset \hat{F}_{\mathcal{T}}$ (derived), to maximize the modeling accuracy for a given algorithm, L and measure, m.

$$F^* = \arg\max_{F_1, F_2} A_L^m(F_1 \cup F_2, y) \tag{9.1}$$

9.3 A Few Simple Approaches

One way of constructing new features is to simply apply all transformations to the given data and sum all the resulting datasets. This leads to the generation of a large number of features, a few of which might be useful with respect to the given target. However, training a model over such large feature set is computationally inefficient and also leads to overfitting. It is possible to reduce the number of features through a feature selector, retaining only a relevant subset of features. This process is illustrated in Figure 9.3(a). This technique is easy to implement and and effective in finding features that can be generated from a single layer of transformations on the given features. However, it lacks the capability to generate features from the composition of different transformations, often limiting the scope of the feature space it can discover. Note that its not feasible to run this method recursively because of the magnitude of feature expansion. The feature subset selection algorithms can be a performance bottleneck because of their super-linear complexity in the number of features. Note that this technique doesn't explicitly involve training and evaluations; however, they may be performed within the feature selection step. This approach is suggested as a part of Data Science Machine (DSM) [10] and OneButton Machine [15]. We refer to this as the *expansion-reduction* approach.

A contrasting approach to above is to generate one new feature at a time followed by training and evaluation to decide if the new feature is worth keeping or not. While this method is more scalable that the expansion-reduction approach, it is also slower because it involves model trainings and evaluations, over the entire space of features that can be generated. In practice, this method is also only feasible without deep compositions of transforms because of the expensive nature of exploration. ExploreKit [11] describes one such method, where a greedy heuristic logic for feature priortization is used. We will call this category as the *evolution-centric* approach. It is illustrated in Figure 9.3(b).



(a) Expansion-reduction. (b) Evolution-centric.

FIGURE 9.3: Two basic methods for feature engineering.

The two contrasting approaches discussed in the previous section pose their own unique performance challenges. The expansion-reduction method has scalability problems due to the dependence on a feature subset selection module running for a large number of generated features. Whereas the evolution-centric approach is fairly time consuming due to the training and evaluation with respect to each new feature. In a way, the two approaches are the opposite of each other and go to different extremes, which also cause each of them to be inefficient in its own way. Recall the example of BMI which is derived through a composition of two basic transformation functions (square and division) on two measured quantities. Either of the two approaches are unlikely to discover BMI. In the next section, we will explore a middle-path, where the newly generated features are batched into groups by applying set transformations on entire datasets.

9.4 Hierarchical Exploration of Feature Transformations

So far we have discussed two algorithmic approaches to feature engineering. In summary, expansion-reduction works by generating a large number of new features, followed by pruning the undesired ones. On the other hand, the evolution-centric method evaluates the addition of one feature at a time. The two approaches stand in contrast to each other in the quantity and timing of generation of new features. Both face performance bottlenecks because of the extremities of their approaches. Additionally, the given approaches do not clearly embrace the composition of transforms. Composition is essential for the discovery of complex relationships. In this section, we discuss an approach that

overcomes some of these limitations through *batching* of new feature generation and evaluation. It also performs a *hierarchical composition* of transforms in a performance driven manner.

The batching is performed per transformation. At each step, one transformation is applied to a dataset (recall the set level transformations with capital letters), generating a set of new features. At each step, the resulting new dataset is evaluated for accuracy. This is performed recursively, forming a hierarchical structure. The batching of feature generation per transformation is scalable and efficient. It also provides an abstraction for measuring the effectiveness of each transformation on the data. The difference in model accuracy upon applying a transformation is averaged over all instances of its application for the given problem until that moment. Those performance numbers are then used to guide the exploration process – to make the decision of which transformation to apply next, to which version of the dataset. The hierarchical organization is a directed acyclic graph (DAG), known as a Transformation Graph[13]. It is a general framework for performance based feature engineering. The approaches discussed before this can be expressed as specific formulations of this approach. Further in this section, we formally define the anatomy of the transformation graph, followed by strategies to explore one.

9.4.1 Transformation Graph

A *Transformation Graph*, G, for a given dataset, D_0, and a finite set of transformations, $\mathcal{T}$, is a directed acyclic graph in which: each node corresponds to a either D_0 or a dataset derived from D_0 using a transformation path. Hence, every node's dataset contains the same target and row count as D. The nodes are divided into three categories: (a) The start or the root node, D_0 corresponding to the given dataset; (b) Hierarchical nodes, D_i, where $i > 0$, which have one incoming parent node D_j, $j > i$, and the connecting edge from D_j to D_i corresponds to a transformation $T \in \mathcal{T}$ (including feature selection), i.e., $D_j = T(D_i)$. The direction of an edge represents the application of the transformation from source to a target dataset or node; (c) Sum nodes, $D^+_{i,j} = D_i + D_j$, a result of a dataset sum such that $i \neq j$. Height (h) of the transformation graph is the maximum unweighted distance between the root and any other node. The operator $\theta(G)$ signifies all nodes of graph G. Also, $\lambda(D_i, D_j)$ signifies the transformation T, such that its application on D_i created D_j as its child. A transformation graph is illustrated in Figure 9.4. The best known solution through a transformation graph is the node with the greatest accuracy: $\arg\max_{D_i} A(D_i)$. A complete transformation graph always contain a global solution to the problem.

Any complete transformation graph is unbounded for a non-empty transformation set. A constrained (with bounded height, h) but complete transformation graph for t transformations contains $t^{h+1} - 2$ hierarchical nodes, and $\frac{(t^{h+1}-1)\times(t^{h+1}-2)}{2}$ sum nodes. It can be seen that for even a height bounded tree with a modest number of transformations, the verification of accuracies

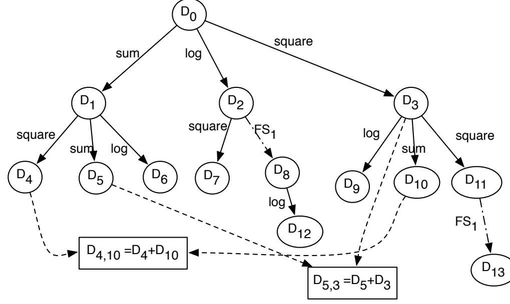


FIGURE 9.4: Example of a *Transformation Graph*, a directed acyclic graph. The start node D_0 corresponds to the given dataset. The hierarchical nodes are circular and the sum nodes are rectangular. Here we can see three transformations: *log*, *sum*, and *square*, as well as a feature selection operator FS_1.

across the tree is combinatorially large. Therefore, we adopt a *performance guided exploration* strategy to explore only a tiny subset of the graph which is most likely to contain the required solution, avoiding other nodes. The algorithm works under a budget constraint.

9.4.2 Transformation Graph Exploration

Exhaustive exploration of a transformation graph is not an option, given its massive potential size. For instance, with 20 transformations and a height = 5, the complete graph contains about 3.2 million nodes; an exhaustive search would imply as many model training and testing iterations. On the other hand, there is no known property that allows us to deterministically verify the optimal solution in a subset of the trials. Hence, the focus of this work is to find a performance driven exploration policy or strategy, which *maximizes expected gain in accuracy* in a *limited time budget*.

Algorithm 1 General Transformation Graph Exploration

Input: Dataset D_0, Budget B_{max}

1: Initialize G_0 with root D_0
2: **while** $i < B_{max}$ **do**
3: $\quad \mathcal{N} \leftarrow \theta(G_i)$
4: $\quad b_{ratio} \leftarrow \frac{i}{B_{max}}$
5: $\quad n^*, t^* \leftarrow \arg\max_{n,t \nexists n' \forall t=\lambda(n,n')} R(G_i, n, t, b_{ratio})$
6: $\quad G_{i+i} \leftarrow$ Apply t^* to n^* in G_i
7: $\quad i \leftarrow i + 1$
8: **end while**

Output: $\underset{D}{\text{argmax}}\ A(\theta(G_i))$

Algorithm 1 outlines a general methodology for exploration. At each step, an estimated reward from each possible move, $R(G_i, n, t, \frac{i}{B_{max}})$, is used to rank the options of actions available at each given state of the transformation graph $G_i, \forall i \in [0, B_{max})$, where B_{max} is the total allocated budget in number of steps. The budget can be considered in terms of any quantity that is monotonically increasing in i, such as time elapsed; for simplicity, we work with "number of steps". Note that the algorithm allows for plugging-in of different exploration strategies, through the definition of the function $R(\dots)$. Any such definition is a function of four basic parameters: (1) current global state of the graph, (2) which transform is being characterized for application, (3) which node is it being considered to apply on, and (4) how much budget is remaining. The following is a non-exhaustive list of influential factors (attributes of one or more of the four parameters) in designing an exploration strategy:

1. Node n's Accuracy: Higher accuracy of a node incentives further exploration from that node, compared to others.
2. Transformation, t's average or max accuracy improvement until G_i.
3. Number of times transform t has already been used in the path from root node to n. A high or even non-zero number weakens the case for an application of t on n.
4. Accuracy gain for node n (from its parent) and the accuracy gain for n's parent. This tests whether n's cumulative gains are recent or not.
5. Node Depth: A higher value is considered as a sign of relative complexity of the transformation sequence.
6. The fraction of budget exhausted till G_i.
7. Ratio of feature counts in n to the original dataset: This indicates how bloated n is, in comparison to the original dataset.
8. Is the transformation a feature selector (reduces feature count)?
9. Whether the dataset contain (a) numerical features; or (b) date-time features; or (c) string features?

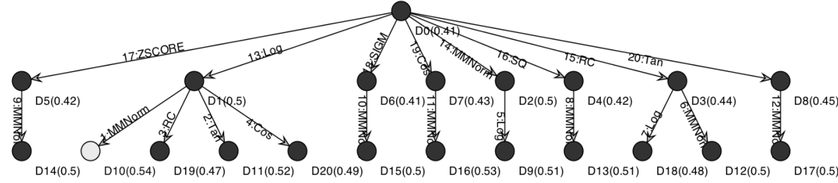
The exploration strategy essentially translates to the design of the reward estimation function, $R(\dots)$. Some of the handcrafted graph traversal strategies by Cognito [14] are describes as follows. In a *depth-first* strategy, the emphasis is on exploring further from the node with the highest accuracy, until saturation or decrease in accuracy is noticed. It is best utilized for finding consolidating on an already found-solution in a somewhat limited budget. However, it can get stuck in local, deep areas of the graph without exploring other simpler choices. In other words, it lacks the exploration aspect and such a policy may take a long time to stumble upon a simple transformation with

high reward. A *breadth-first* strategy is primarily focused on exploring the less explored subtrees of the hierarchical portion of the transformation graph. Other factors are secondary influencers – such as transformation performance, parent node's accuracy and child node's prospective accuracy. This strategy is good for discovering single (or a small sequence) of highly rewarding transforms. However, it performs poorly in consolidating benefits into a single chain of of large number of transforms. A *global* strategy is derived from a mix of the depth and breadth oriented policies. It works with first exploring the breadth, followed by a more concentrated exploration of promising depths based upon the initial phase. Figure 9.4.2 illustrates the breadth-first, depth-first and one of the RL-based strategies (Section 9.5) for OpenML_618 dataset ($B_{max} = 20$, $h_{max} = 5$), with sum-nodes disabled (for visual clarity). In general, it is hard to manually encode all rules that best capture the optimal intent for different situations – at different values of remaining budget, average performance of different transformations and their combinations for a given dataset, and so on. In the next section, we will discuss how to empirically optimize that.

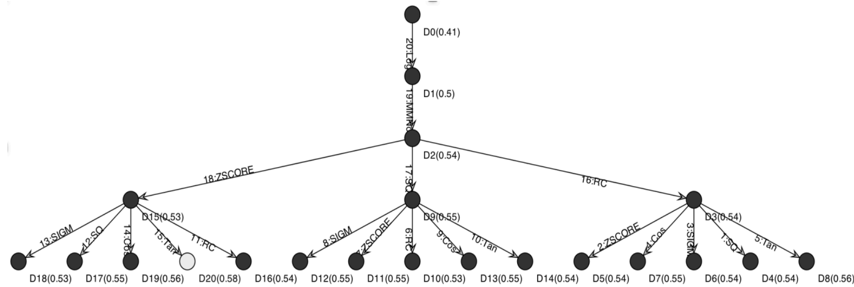
9.5 Learning Optimal Traversal Policy

In the previous section, we discussed an algorithmic approach for a performance-based exploration for feature engineering. It was based upon the exploration of a hierarchical set-up of the transformation functions in the form of a transformation graph (a directed acyclic graph). We discussed different heuristics to form exploration strategies based on an understanding of the manual trial and error process. While it helps achieve good results for feature engineering without human intervention, there is a considerable scope for improving the strategy beyond the handcrafted rules.

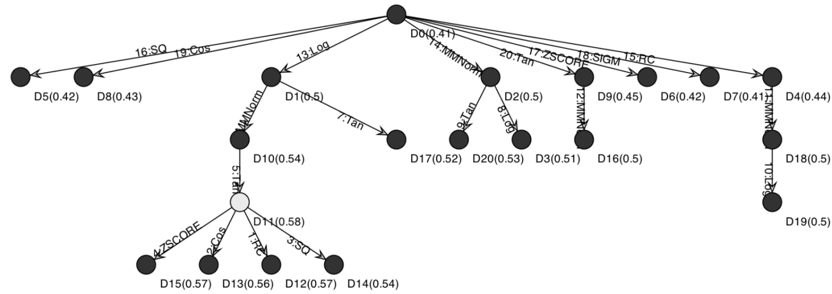
In this section, we describe a method to improve the exploration strategy from experience. Instead of relying on "human experts" to encode heuristics, we rely on empirical observations and perform *reinforcement learning* to optimize the strategy. Consider a feature engineering *agent* that is continuously monitoring the impact of each transformation applied (call it *action*) on a given transformation graph (*state*) and resulting improvement in performance (*reward*). The goal of reinforcement learning here is for the agent to learn a strategy that the optimizes the final or cumulative reward ($A^* - A(D_0)$) for a given dataset in a specified time budget. Similar to the handcrafted strategies discussed previously, the learned strategy can also be thought of as an *action-utility* function to satisfy the expected reward function, R(...) in Algorithm 1. This simply means the association of any action with a scalar utility or expected reward value. Reinforcement learning is helpful in learning such an action-utility function based upon observing immediate rewards from actions, in the process of optimizing the final or cumulative reward. A



(a) Bredth-first exploration.



(b) Depth-first exploration.



(c) RL-based exploration.

FIGURE 9.5: Illustration of different exploration policies on dataset: https://www.openml.org/d/618 dataset. In $B_{max} = 20$ iteration limit, RL and DF both find the best performance of 0.58 ($1 - Rel.Abs.Error$), while BF finds only 0.54. RL takes 11 iterations, while DF takes 20 iterations.

tutorial on reinforcement learning is beyond the scope of this chapter, but we encourage the interested reader to refer to Sutton and Barto [26] for a general understanding of the topic. In a nutshell, reinforcement learning is an area of machine learning concerned with training an agent to perform optimal actions in an environment in order to maximize a notion of cumulative reward. It relies on a reward signal for each action taken in order to guide the agent's behavior, which is different than the supervised learning paradigm, where we are given the ground truth to train the system. Here, we specifically discuss a particular instance of Q-learning with function approximation method as

discussed by Khurana et al. [13]. Q-value learning is a particular kind of reinforcement learning that is useful in the absence of an explicit model of the environment, which in this case translates to the behavior of the learning algorithm. An approximation function is suitable due to the large number of states (recall, millions of nodes in a graph with small depth) for which it is infeasible to learn state-action transitions explicitly. This style of work, where machine learning is aided by the use of other machine learning techniques, is referred to as "learning to learn" or *meta-learning*.

9.5.1 Feature Exploration through Reinforcement Learning

Consider the graph exploration process as a *Markov Decision Process (MDP)* where the *state* at step i is a combination of two components: (a) transformation graph after i node additions, G_i (G_0 consists of the root node corresponding to the given dataset. G_i contains i nodes); (b) the remaining budge at step i, i.e., $b_{ratio} = \frac{i}{B_{max}}$. Let the entire set of states be S. On the other hand, an *action* at step i is a pair of existing tree node and transformation that hasn't already been applied to it, i.e., $< n, t >$ where $n \in \theta(G_t)$, $t \in T$ and $\nexists n \in G_i \forall \lambda(n, n') = t$. Let the entire set of actions be C. A policy, $\Pi : S \rightarrow C$, determines which action is taken given a state. Note that the objective here is to learn the optimal policy (exploration strategy) by learning the action-value function, which is elaborated later.

The described formulation uniquely identifies each state. Considering the "remaining budget" as factor in the state of the MDP helps address the *runtime exploration versus exploitation* trade-off for a given dataset. Note that this runtime explore/exploit trade-off is not identical to the commonly referred trade-off in RL training in context of selecting actions to balance reward and not getting stuck in a local optimum.

At step i, the occurrence of an action results in a new node, n_i, and hence a new dataset on which a model is trained and tested, and its accuracy $A(n_i)$ is obtained. To each step, we attribute an immediate scalar reward:

$$r_i = \max_{n' \in \theta(G_{i+1})} A(n') - \max_{n \in \theta(G_i)} A(n)$$

with $r_0 = 0$, by definition. The cumulative reward over time from state s_i onwards is defined as:

$$R(s_i) = \sum_{j=0}^{B_{max}} \gamma^i . r_{i+j}$$

where $\gamma \in [0, 1)$ is a discount factor, which prioritizes early rewards over the later ones.

We use Q-learning [28] with function approximation to learn the action-value Q-function. For each state, $s \in S$ and action, $c \in C$, Q-function with respect to policy Π is defined as:

$$Q(s, c) = r(s, c) + \gamma R^{\Pi}(\delta(s, c))$$

where $\delta : S \times C \rightarrow S$ is a hypothetical transition function, and $R^{\Pi}(s)$ is the cumulative reward following state s. The optimal policy is achieved as:

$$\Pi^*(s) = \arg\max_c [Q(s,c)] \tag{9.2}$$

However, given the size of S, it is infeasible to learn Q-function directly. Instead, a linear approximation the Q-function is used as follows:

$$Q(s,c) = w^c.f(s) \tag{9.3}$$

where w^c is a weight vector for action c and $f(s) = f(g,n,t,b)$ is a vector of the state characteristics described in the previous section and the remaining budget ratio. Therefore, we approximate the Q-functions with linear combinations of the characteristics of a state of the MDP. Note that, in the heuristic rule-based strategies described in Section 9.4.2, we used a subset of these state characteristics, in a self-conceived manner. However, in the ML based approach here, we select the entire set of characteristics and empirically determine the appropriate weights of those characteristics (for different actions). Hence, this approach generalizes the handcrafted approaches.

The update rule for w_c is as follows:

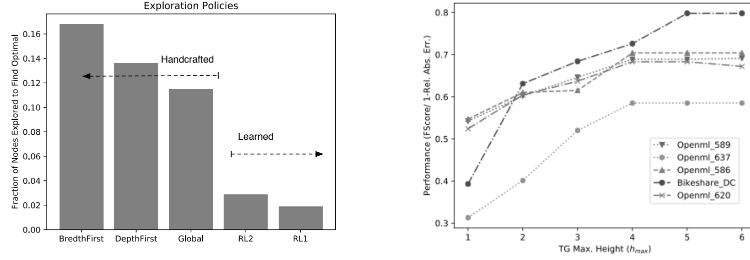
$$w^{c_j} \leftarrow w^{c_j} + \alpha.(r_j + \gamma. \max_{n',t'} Q(g',c') - Q(g,c)).f(g,b) \tag{9.4}$$

where g' is the state of the graph at step $j+1$, and α is the learning rate parameter. The proof follows from [9].

A variation of the linear approximation where the coefficient vector w is independent of the action c, is as follows:

$$Q(s,c) = w.f(s) \tag{9.5}$$

This method reduces the space of coefficients to be learnt by a factor of c, and makes it faster to learn the weights. It is important to note that the Q-function in this case is still not independent of the action c, as one of the factors in $f(s)$ or $f(g,n,t,b)$ is actually the average immediate reward for the transform for the present dataset. Hence, Equation 9.5 based approximation still distinguishes between various actions (t) based on their performance in the transformation graph exploration so far; however, it does not learn a bias for different transformations in general and based on the feature types (factor #9). We refer to this type of strategy as RL_2. In our experiments RL_2 efficiency is somewhat inferior to the strategy to the strategy learned with Equation 9.3, which we refer to as RL_1. However, RL_2 can be learned from fewer examples compared to RL_1, due to the former's smaller space of parameters. In Figure 9.6(a), we see that on an average for 10 datasets, the RL-based strategies are 4-8 times more efficient than any handcrafted strategy (*breadth-first, depth-first* and *global* as described in [14]), in finding the optimal dataset in a given graph with 6 transformations and bounded height, $h_{max} = 4$.



(a) Comparing different exploration policies by an average of nodes explored (in a constrained graph) to find the optimal solution.

(b) Performance of RL_1 exploration on various datasets with varying h_{max}. $h_{max} = 1$ is the base accuracy.

FIGURE 9.6: Evaluating the performance for hierarchical exploration.

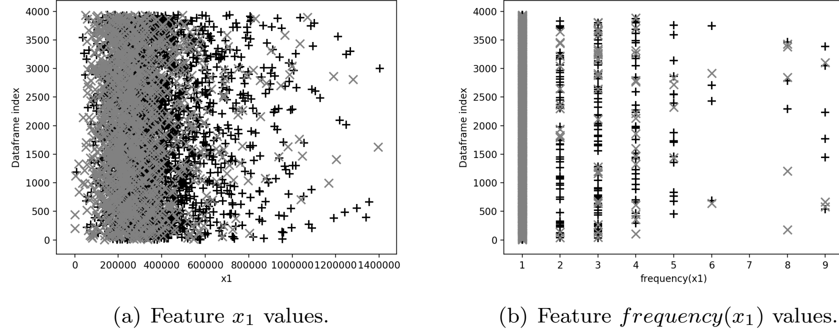
For training, Khurana at al. [13] used 48 datasets (not overlapping with test datasets) to select training examples using different values for maximum budget, $B_{max} \in \{25, 50, 75, 100, 150, 200, 300, 500\}$ with each dataset, in a random order. The discount factor, $\gamma = 0.99$, and learning rate parameter, $\alpha = 0.05$. The weight vectors, w^c or w, each of size 12, were initialized with 1's. The training example steps were drawn randomly with the probability $\epsilon = 0.15$ and the current policy with probability $1 - \epsilon$.

9.6 Finding Effective Features without Model Training

So far, we have discussed approaches that rely on evaluation of generated features either directly through model construction and testing, or indirectly through feature selection. These tasks are computationally expensive. In this section, we shift the discussion to a paradigm without model construction and evaluation. Consider a binary classification example where one of the features is plotted in Figure 9.7(a). The high degree of overlap between the two classes suggests that this feature is not quite helpful for classification. Upon transformation with a frequency function, the distinction between points from the two classes is more prominent, as can be seen on the right. Hence, without model training and evaluation, we can suggest that the particular transformation has generated additional value for classification with respect to the base feature. Generally, in the presence of a set of other features F, we can only say that a feature f_2 is more suitable than f_1, , if,

$$\Pr(y|f_2, F) > \Pr(y|f_2, F) \tag{9.6}$$

Determining the validity of Inequation 9.6 is as good as training two models



(a) Feature x_1 values. (b) Feature $frequency(x_1)$ values.

FIGURE 9.7: Scatterplots juxtaposing values of an original and a transformed feature for several data instances ("dataframe index"). The transformed feature separates the two classes ('x' / '+') better than the original.

with feature sets $f_1 + F$ and $f_2 + F$, respectively. In practice however, it is still viable to consider only the independent impact of a derived features with respect to its base feature. This is because amongst a vast pool of features that can be derived, only a small fraction ever up being beneficial. For a base feature f_1, a transformation, t, and a derived feature, $t(f_1)$, the following condition makes $t(f_1)$ a strong candidate to add value to the problem of predicting y:

$$\Pr(y|t(f_1), f_1) > \Pr(y|f_1) \tag{9.7}$$

A more strict condition than Equation 9.7, but one that is easier to evaluate through proxy functions is the following:

$$\Pr(y|t(f_1)) > \Pr(y|f_1) \tag{9.8}$$

Consider the case of binary classification. Measuring the overlap of two sets of a feature belonging to different classes provides a reasonable measure of effectiveness of that feature itself in building a classifier. Lesser the overlap, the better. The degree of lack of overlap can be measured by the magnitude of divergence in the probability distribution functions (PDFs) of the two classes, say $\omega(c_1, c_2)$. If the application of a transformation reduces ω, it is a positive signal to embrace the new feature. One such measure is the symmetric-KL-divergence. It is described below for continuous distributions of the two classes, $c_1(x)$ and $c_2(x)$ for feature, f. There also exists a corresponding expression for discrete distributions.

$$\omega_{KL}^{f}(c_1, c_2) = \int_{-\infty}^{\infty} (c_1(x) - c_2(x)) \log \frac{c_1(x)}{c_2(x)} dx \tag{9.9}$$

If $\omega^{f_2} > \omega^{f_1}$, where $f_2 = t(f_1)$ for a base feature f_1 – we can infer that

$f_2 = t(f_1)$ is a potentially valuable instance of transformation and f_2 should be retained. It is important to note two important points regarding the choice of the measure. First, metrics such as mutual information are not suitable. For our purpose, even identically shaped distributions without a significant overlap are good cases and mutual information does not convey that difference. Secondly, the required measure need not strictly be a metric. This concept extends well to multi-class problems. In case of regression, a measure of correlation between the feature and the target mirrors the objective appropriately.

This approach is efficient because the measures of similarity between PDFs can be approximated more efficiently than training models on the entire data. However, it still needs an enumeration of various feature-transformation choices. While the computed PDF for a feature can be cached (specifically for base features), creating the PDF for each new feature ($|\mathcal{T}| \times |F|$), and computing ω is still quite a bit of work.

9.6.1 Learning to Predict Useful Transformations

We now discuss an approach to predict ω using supervised machine learning instead of computing it. It is based on *learning* the patterns between the distributions of feature vectors, the target vector and corresponding utility of transformations. Once learned, those patterns are employed for predicting useful transformations for any previously unseen dataset. Its prominent advantage is that it is not dependent on the expensive task of model training and evaluation. It does not even depend on finding the improvement in divergence of PDFs of different classes upon applying a transformation. Instead, it predicts the improvement based on a trained model. Hence, it is much faster at runtime compared to any other method discussed so far. It can also be combined with any of the previously described approaches to prioritize their transformation application in accordance with the predictions. This style of making transformation predictions for a dataset based upon past experience is somewhat analogous to the hunch or intuition used by a data scientist in manual feature engineering.

Nargesian et al.[20] train a set of Multi-Layer Perceptrons (MLP), one for each transformation $t \in \mathcal{T}$. For every given feature-target pair, an MLP learns the impact of its corresponding transformation on the specified prediction task. It generalizes this knowledge across all training examples. A training instance consists of an alternative representation of the feature's PDF as input; the output is a binary value – whether the transformation is useful for accuracy improvement of the model or not. Hundreds of thousands of training examples are used, thanks to the vast array of open dataset repositories for supervised learning problems. Notice that the training data for the supervised meta-learning problem is generated automatically.

Let R_f be the alternate representation for a feature f. Recommending a transformation for f involves applying all $|\mathcal{T}|$ MLPs on R_f. If the highest confidence score obtained from the classifiers that returned a positive output

is above a given threshold, the corresponding transformation is recommended for application on feature f. Let $G_k(R_f)$ be the confidence score of the MLP corresponding to transformation t_k, and γ is the threshold for confidence scores which we determined empirically. LFE recommends the transformation t_c, for feature f, as follows:

$$c = \arg\max_k G_k(R_f)$$
$$\text{recommend} : \begin{cases} t_c, & \text{if } G_c(R_f) > \gamma \\ \text{none}, & \text{otherwise} \end{cases} \tag{9.10}$$

The alternative PDF representation is called a *Quantile Sketch Array (QSA)*. It represents feature f in a dataset with k classes as follows:

$$R_f = \left[Q_f^{(1)}; Q_f^{(2)}; \ldots; Q_f^{(k)}\right] \tag{9.11}$$

where $Q_f^{(i)}$ is a fixed-sized representation of values in f that are associated with class i. QSA uses *quantile* data sketch [27] to represent feature values associated with a class label. It is a non-parametric representation that enables characterizing the feature PDFs. QSA is similar to a cumulative histogram, where data is summarized into a small number of buckets. Several other approaches for the alternate representation have been tried and QSA has proved to be the most effective for this problem [20].

Let $\mathcal{V}_k$ be the bag of values in a feature, f, that correspond to label c_k and $Q_f^{(i)}$ is the quantile sketch of $\mathcal{V}_k$. First, these values are scaled to a predefined range $[lb, ub]$. To generate $Q_f^{(i)}$, all values in $\mathcal{V}_k$ are bucketing into a set of bins. Given a fixed number of bins, r, the range $[lb, ub]$ is partitioned into r disjoint bins of uniform width $w = \frac{ub-lb}{r}$. The range of the bin b_j ($j \in \{0, 1, \ldots r-1\}$) is $[lb + j * w, lb + (j+1) * w)$. $P(b_j)$ signifies the number of feature values bucketed in b_j. And, $I(b_j) = \frac{P(b_j)}{\sum_{0 \le m < r} P(b_m)}$ is the normalized value of $P(b_j)$ across all bins.

The training samples for transformation MLP classifiers are generated using several classification datasets. For each dataset, each numerical features, f, is considered and a model is trained using a learning algorithm, L. For each MLP, the corresponding transform t is applied to f and a new model is trained with f and $t(f)$. If the transformation leads to an improvement above a certain threshold, $A_L(\{f, t(f)\}, y) - A_L(\{f\}, y) > \phi$, R_f is considered a *positive* training example; otherwise a *negative* training example.

For a k-class problem, and while using b bins for each quantile data sketch, R_f is a vector of size $k \times b$. For an MLP with one hidden layer having h units, the probability of transformation t being a useful transformation or not for feature f is computed as:

$$\begin{aligned}&[p_{t \text{ is useful}}(f), p_{t \text{ is not useful}}(f)] = \\ &\sigma_2(\mathbf{b^{(2)}} + \mathbf{W^{(2)}}(\sigma_1(\mathbf{b^{(1)}} + \mathbf{W^{(1)}}[Q_f^{(1)}; \ldots; Q_f^{(k)}])))\end{aligned} \tag{9.12}$$

Here, $\mathbf{W^{(1)}}$ and $\mathbf{W^{(2)}}$ are weight matrices, $\mathbf{b^{(1)}}$ and $\mathbf{b^{(1)}}$ are bias vectors. σ_1 and σ_2 are softmax and rectified linear unit (ReLU) functions, respectively. Stochastic Gradient Descent is used to train transformation MLPs. Overfitting should be prevented using regularization and drop-out [24].

At runtime, when a new dataset is presented, computing the QSA and scoring on the MLPs is computationally cheap. Compared to evaluation-based techniques with a large turn around time, prediction-based techniques can give quick insights to a data scientist. Additionally, their results can be used as an initial bias for any of the exploration-based techniques for faster search. The gains reported by this technique are usually better than expansion-reduction style of feature engineering [20], but less than a well-tuned transformation graph exploration system.

9.7 Miscellenious

9.7.1 Other Related Work

The techniques presented in this chapter are representative of the state-of-the-art for automated feature engineering for predictive modeling. There is an additional body of valuable work which should also be of interest for researchers and graduate students working on this topic. We summarize some of the relevant work in this section.

FICUS [18] performs a beam search over the space of possible features, constructing new features by applying "constructor functions" (e.g. inserting an original feature into a composition of transformations). FICUS's search for better features is guided by heuristic measures based on information gain in a decision tree, and other surrogate measures of performance. FICUS is more general than a number of less recent approaches [21, 1, 29, 19, 8].

Fan et al. [4] propose FCTree uses a decision tree to partition the data using both original and constructed features as splitting points (nodes in the tree). Similar to FICUS [18], FCTree uses surrogate tree-based information-theoretic criteria to guide the search, as opposed to the true prediction performance. FCTree is capable of generating only simple features, and is not capable of composing transformations, i.e. it is search in a smaller space than our approach. They also propose a weight update mechanism that helps identify good transformations for a dataset, such that they are used more frequently. FEADIS [3] relies on a combination of random feature generation and feature selection. It adds constructed features greedily, and as such requires many expensive performance evaluations.

Certain machine learning methods perform some level of feature engineering implicitly. A recent survey on the topic appears can be found here [25]. Dimensionality reduction methods such as Principal Component Analysis (PCA)

and its non-linear variants (Kernel PCA) [6] aim at mapping the input dataset into a lower-dimensional space with fewer features. Such methods are also known as *embedding* methods [25]. Kernel methods [22] such as Support Vector Machines (SVM) are a class of learning algorithms that use kernel functions to implicitly map the input feature space into a higher-dimensional space.

9.7.2 Research Opportunities

One interesting direction is to view the problem as a hyper-parameter optimization [2]. Each transformation option corresponds to a hyper-parameter and one searches for the hyper-parameter setting that results in the best improvement of predictive performance. For instance, in [23] a genetic algorithm was used to determine a suitable transformation for a given data set. Similarly, in the context of automated ML pipeline configuration (e.g., feature selection and model), the work presented in [5] employs Bayesian optimization to determine a suitable pipeline. While the approach in [23] is limited to determining single transformations and does not search for a sequence of transformations, black-box optimization strategies [17] have to our knowledge not been applied to generate novel features based on compositions. A fertile area for improvement is to combine different known methods for feature engineering into one. For instance, using prediction-based techniques to provide quick insights to exploration-based techniques can boost the overall efficiency of the process [12]. Finally, extending the analysis and solutions presented in Section 9.6 from a single base feature to multiple ones and perform a joint-probabilistic analysis is valuable.

9.7.3 Resources

Please refer to an online addendum to this chapter on GitHub `https://github.com/uk2911/FEChapterExtended`. It contains resources to automated feature engineering tools that are described in this chapter. There are demonstrations on various real datasets, including videos and IPython notebooks. It also contains hints on efficiently implementing your own feature engineering program.

Acknowledgement: Thanks to Horst Samulowitz, Fatemeh Nargesian, Elias Khalil, Deepak Turaga, and Tejaswini Pedapati for joint research on different problems which are described in this chapter. Thanks to Biplav Srivastava for helpful discussions on the process of writing the chapter.

Bibliography

[1] Giulia Bagallo and David Haussler. Boolean feature discovery in empirical learning. *Machine learning*, 5(1):71–99, 1990.

[2] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011.

[3] Ofer Dor and Yoram Reich. Strengthening learning algorithms by feature discovery. *Information Sciences*, 2012.

[4] Wei Fan, Erheng Zhong, Jing Peng, Olivier Verscheure, Kun Zhang, Jiangtao Ren, Rong Yan, and Qiang Yang. Generalized and heuristic-free feature construction for improved accuracy. pages 629–640, 2010.

[5] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. *NIPS*, 2015.

[6] Imola K Fodor. A survey of dimension reduction techniques, 2002.

[7] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[8] Yuh-Jyh Hu and Dennis Kibler. Generation of attributes for learning algorithms. *AAAI*, 1996.

[9] Marina Irodova and Robert H Sloan. Reinforcement learning and function approximation. In *FLAIRS Conference*, pages 455–460, 2005.

[10] James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. *IEEE Data Science and Advanced Analytics*, pages 1–10, 2015.

[11] Gilad Katz, Eui Chul, Richard Shin, and Dawn Song. Explorekit: Automatic feature generation and selection. In *IEEE ICDM*, pages 979–984, 2016.

[12] Udayan Khurana, Fatemeh Nargesian, Horst Samulowitz, Elias Khalil, and Deepak Turaga. Automating feature engineering. In *Artificial Intelligence for Data Science (NIPS workshop)*, 2016.

[13] Udayan Khurana, Horst Samulowitz, and Deepak Turaga. Feature engineering for predictive modeling using reinforcement learning. *arXiv preprint arXiv:1709.07150*, 2017.

[14] Udayan Khurana, Deepak Turaga, Horst Samulowitz, and Srinivasan Parthasarathy. Cognito: Automated feature engineering for supervised learning. In *IEEE ICDM*, 2016.

[15] Hoang Thanh Lam, Johann-Michael Thiebaut, Mathieu Sinn, Bei Chen, Tiep Mai, and Oznur Alkan. One button machine for automating feature engineering in relational databases. *arXiv preprint arXiv:1706.00327*, 2017.

[16] Jundong Li, Kewei Cheng, Suhang Wang, Fred Morstatter, Trevino Robert, Jiliang Tang, and Huan Liu. Feature selection: A data perspective. *arXiv:1601.07996*, 2016.

[17] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR*, abs/1603.06560, 2016.

[18] Shaul Markovitch and Dan Rosenstein. Feature generation using general constructor functions. *Machine Learning*, 2002.

[19] Christopher J Matheus and Larry A Rendell. Constructive induction on decision trees. *IJCAI*, 1989.

[20] Fatemeh Nargesian, Horst Samulowitz, Udayan Khurana, Elias B. Khalil, and Deepak Turaga. Learning feature engineering for classification. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 2529–2535, 2017.

[21] Harish Ragavan, Larry Rendell, Michael Shaw, and Antoinette Tessmer. Complex concept acquisition through directed search and feature caching. *IJCAI*, 1993.

[22] John Shawe-Taylor and Nello Cristianini. *Kernel methods for pattern analysis*. Cambridge university press, 2004.

[23] Matthew G. Smith and Larry Bull. *Feature Construction and Selection Using Genetic Programming and a Genetic Algorithm*, pages 229–237. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[24] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning*, 15(1):1929–1958, 2014.

[25] Dmitry Storcheus, Afshin Rostamizadeh, and Sanjiv Kumar. A survey of modern questions and challenges in feature extraction. *Proceedings of The 1st International Workshop on Feature Extraction, NIPS*, 2015.

[26] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[27] Lu Wang, Ge Luo, Ke Yi, and Graham Cormode. Quantiles over data streams: An experimental study. *SIGMOD*, pages 737–748, 2013.

[28] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[29] Der-Shung Yang, Larry Rendell, and G Blix. Fringe-like feature construction: A comparative study and a unifying scheme. *ICML*, 1991.