# Chapter 1
# Learning from Limited Data in VLSI CAD

Li-C. Wang

**Abstract** Applying machine learning to analyze data from design and test flows has received growing interests in recent years. In some applications, data can be limited and the core of analytics becomes a feature search problem. In this context, the chapter explains the challenges for adopting a traditional machine learning problem formulation view. An adjusted machine learning view is suggested where learning from limited data is treated as an iterative feature search process. The theoretical and practical considerations for implementing such a search process are discussed.

## 1.1 Introduction

Applying machine learning tools to analyze data collected from design and test processes can encounter different types of learning problems. One common type of analytics is based on dividing data samples into two classes. For example, the analytic begins with $m$ samples where there are $m_p$ *positive samples* and the rest are *negative samples*. A set of $n$ features $f_1, \ldots, f_n$ are used to describe each sample. The goal is to learn a model based on those features to differentiate one or more of the positive samples from the negative ones. It is often that $m_p$ is very small and in some cases, even $m_p = 0$. In addition, $m$ is limited as well. A recent paper [1] discusses several applications that involve this type of analytics.

For example, Figure 1.1 illustrates an application in functional verification [2][3]. With a functional verification environment, a *testbench* is instantiated through Constrained Random Test Generation (CRTG) into a set of *functional tests*. The design under verification can be a SoC (System-on-Chip) and each test can be a C program (or a sequence of instructions). Simulating the tests results in *simulation traces*

Li-C. Wang
Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106, USA. e-mail: licwang@ece.ucsb.edu

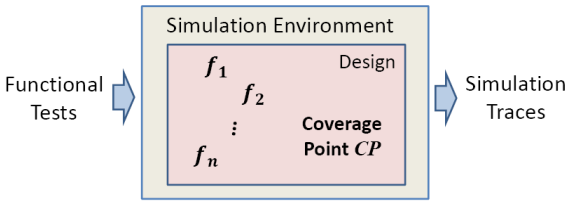


**Fig. 1.1** An application in functional verification

which in this case are the data to be analyzed. The task might concern a *coverage point CP* in the design and the goal is to help improve the coverage of *CP*.

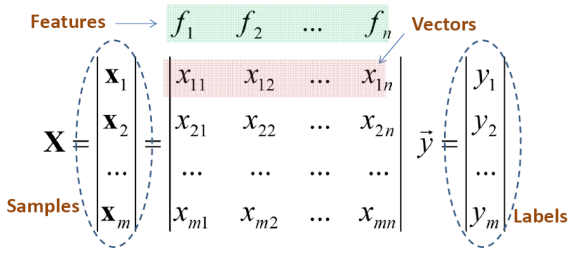


**Fig. 1.2** The format of a dataset to be analyzed

To run an analytic tool, a dataset is formatted as that illustrated in Figure 1.2. First, we need to define what a sample represents. In the context of analyzing simulation traces, a sample can be defined as the activities observed in a simulation cycle. Next, we need to describe the activities with a set of features. Each feature can be defined based on a design signal. Then, each sample can be represented as a vector of digital values. For example, these values can be 0, 1, a rising transition, and a falling transition. Each sample ($\mathbf{x}$) is also associated with a label ($y$) based on the coverage status of *CP*. The coverage status can be from the same cycle or in the next few cycles. If *CP* is covered, the sample is labeled as positive ($+1$). Otherwise, it is labeled as negative ($-1$).

Usually, the coverage point *CP* is of concern because it receives very few or no coverage in the simulation. This means that in the dataset, there are very few or no positive samples. The goal of the analytics can be for deriving a *rule* such that satisfying the rule is likely to achieve coverage of *CP*. Such a rule can be a combination of some feature values. After a rule is reported by an analytic tool, the testbench is modified to produce more tests that satisfy the rule. These tests are simulated and the quality of the rule is observed in the simulation result [2][3].

In addition to functional verification, the paper [1] discusses several other applications that involve this type of analytics. For example, in physical verification, a given layout can be scanned into a sequence of *snippets* [4], i.e. a small window of layout image, as illustrated in Figure 1.3. Each snippet can be described with a set of features, such as attributes related to shape, spacing, materials, etc. A positive sample can be defined as a snippet that potentially causes an issue (i.e. a defect-

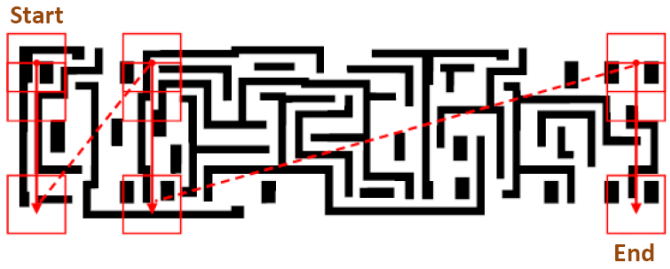


**Fig. 1.3** Raster scan to extract layout snippets

prone spot). Then, the rest are negative samples. Our interest is often in predicting the positive samples. Such an application can also encounter a dataset where there are many more negative samples than positive ones. Moreover, because different positive samples may be due to different reasons, it is likely that for a particular analysis, we are interested in modeling only one or few selected positive samples.

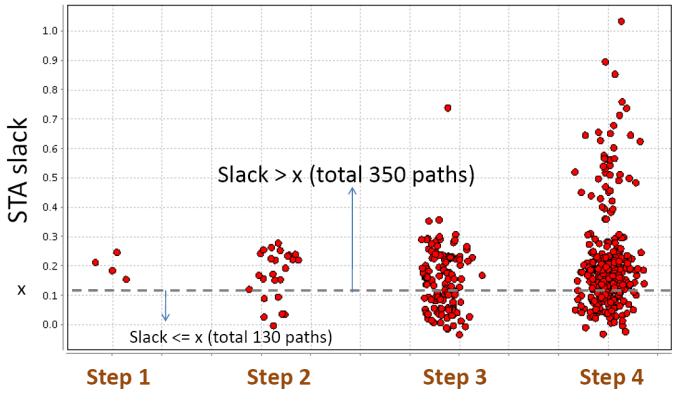


**Fig. 1.4** 350 critical paths not predicted by STA

Another example is timing verification. In this case, samples can be defined as paths (e.g. a path from one flip-flop to another flip-flop). For example, figure 1.4 shows critical paths collected from a silicon experiment [5] and their predicted timing slacks from a Static Timing Analysis (STA) tool. For a path, its actual delays are measured on a set of process cores. The measurement is carried out by *frequency stepping* where step 1 has the lowest frequency. Each dot shown in the plot represents a path. For example, at step 1 there are four silicon-critical paths.

For this particular design, the assumption is that if the timing slack is less than a selected value (denoted as $x$), then the path is reported as a STA-critical path. In the plot, the (normalized) value of $x$ is shown and it can be observed that 350 out of the 480 paths shown in the plot are not reported as STA-critical paths. In fact, based on

the given $x$ value, the STA reports 21,589 STA-critical paths and only 130 of those paths show up in this plot as a silicon-critical path [5].

To understand why a silicon-critical path is not a STA-critical path, one can start the analysis with the first four silicon-critical paths shown on the left of the plot. In this case, the positive samples are those 4 paths. The negative samples could be the remaining 21,459 ($= 21589 - 130$) STA-critical paths which do not show up as a silicon-critical path [5]. To enable the analysis, a set of design-related features are developed for describing a path. These features can be based on, for example, usage of the cells, layout properties, path location, and so on [5].

**Limited number of samples** For the applications discussed above, a given dataset can have very few or no positive samples. Moreover, the number of total samples can also be limited. For example, in functional verification simulation cost is usually a main concern. Hence, the number of cycles simulated in each run is limited. In the timing verification application, the number of paths with measured timing on silicon chips is limited.

In those applications, there can be many choices to derive a feature set. Hence, the underlying problem can be thought of as searching among the large number of features for those few important ones. The difficulty of this search depends on the availability of the positive samples, the total samples, and the number of potential features to search on. For example, to search on a large number of potential features, one might need a large number of samples and at least some positive samples. If obtaining sufficient samples or positive samples is practically prohibited, then the problem can become challenging.

**Feature-based analytics** Analytics involved in the applications discussed above can be called *feature-based analytics* which means the underlying problem is to search for a small combination of features or feature values among a large set of features.

This search is different from traditional *feature selection* studied in machine learning. A traditional feature selection algorithm is entirely data-driven. On the other hand, the core problem faced in feature-based analytics is that the data can be insufficient to determine the importance of some features when they are included in a dataset. If such a dataset is run with a traditional feature selection algorithm the result can be misleading. This is because a typical learning algorithm usually reports some optimized result based on the data but does not report if the data is sufficient or not for learning the result.

Because of the data limitation, feature-based analytics in practice can be more intuitively captured as an iterative feature search process [1]. In each iteration, a subset of features are selected to run an analytic tool with the samples in hand. The result is evaluated until the outcome is satisfied.

**Learning about the features** In general, learning can be seen as achieving two tasks, learning about the features and learning to construct a model based on the features. Traditionally, these two tasks are separated. A learning algorithm focuses on model building and a feature selection algorithm focuses on feature evaluation. Modern machine learning approach such as *deep learning* [6] puts much more emphasis on learning about the features. Similar to that, the emphasis in the applications discussed in this chapter can also be seen as more about learning the features.

In addition to the applications mentioned above, several other analytic tasks involving post-silicon test data can also be seen as learning about the features. For example, one application is to build a model to predict the Fmax (functional maximum frequency) of a chip [7]. In this application, a feature can be based on a flip-flop, a test pattern, a ring oscillator, or a selected critical path. The feature value is a delay structurally measured. The term "structurally" means to measure the delay using some scan structure. The main challenge for this analytic is to decide what features to use. If an effective set of features are used, an accurate model can be built rather easily. Otherwise, it can be quite difficult to learn a good model.

Another example is production yield optimization [8] [9]. In this context, a feature can be an *e-test* characterizing a process parameter on each wafer. A feature can also be based on a measurable property of a production tool. The analytic is to uncover a feature or a combination of features to explain a yield issue. Then, the goal is to improve the yield by adjusting those feature values accordingly [8]. In this context, the learning problem is almost entirely about learning the features.

## 1.2 Iterative Feature Search

The iterative feature search process is illustrated in Figure 1.5. First, it is common that a tool from the machine learning toolbox (e.g. [10]) expects a dataset that is formatted as depicted in Figure 1.2. To produce such a dataset, three steps are performed: sample selection, feature selection, and dataset construction.

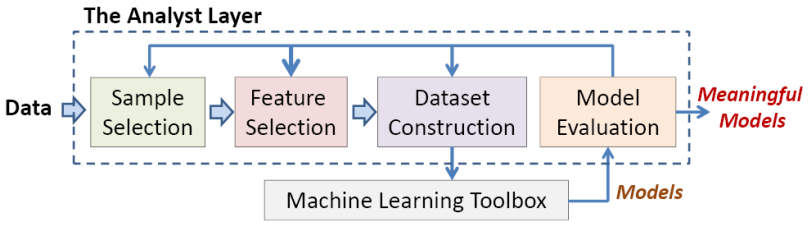


**Fig. 1.5** Iterative feature search process

The sample selection step defines what a sample is and selects a set of $m$ samples $\mathbf{x}_1, \ldots, \mathbf{x}_m$ for the analysis. For example, in functional verification a sample consists

of signal activities in a simulation cycle. In physical verification, a sample is a layout snippet. In timing verification, a sample is a path. For Fmax prediction, a sample is a chip. In yield optimization, a sample can be a chip, a wafer or a lot. If the analysis is based on *supervised learning*, a label value $y_i$ is calculated for each sample $\mathbf{x}_i$. If it is for *unsupervised learning*, no label is required.

Calculating the label for a sample might need a separate analysis itself. For example, in yield analysis a wafer can be classified as good or bad and deciding this binary label can be based on outlier analysis [8] (i.e. a bad wafer has a yield number that is classified as an outlier). Furthermore, it is possible that a particular analysis does not use all samples. For example, after some initial analysis, it might be decided that a subset of samples require a more focused analysis.

An analytic task begins with an initial set of features. This feature set is often developed by consulting a domain expert. Then, in each iteration a subset of features $f_1, \ldots, f_n$ are selected. After this subset is determined, in the dataset construction step for each sample $\mathbf{x}_i$ its feature values $(x_{i1}, \ldots, x_{in})$ are computed. This computation might require running a machine learning tool also. For example, the original values of a feature might be divided into ranges and this division can be based on a clustering algorithm [11].

After a dataset is constructed, running a machine learning tool on the dataset can result in one or more models. For example, a tool can allow setting some parameter values to affect the optimization objective of the learning algorithm. With different parameter values, different models can be obtained. The models then go through a model evaluation step. In this step, the meaningfulness of a model is assessed in the context of the particular application. If model evaluation cannot determine a meaningful model, then a different dataset is required. This invokes a new iteration that can involve redoing one or more of the previous three steps.

### *1.2.1 The need for domain knowledge*

In Figure 1.5, domain knowledge is involved in the Analyst Layer for the dataset preparation and model evaluation. And because of this, learning in view of the figure can be seen as using the data to enhance one's domain knowledge. In other words, Knowledge + Data $\Rightarrow$ Learning [1]. In theory, learning from data would not be possible without any prior knowledge [12]. Hence, the view in Figure 1.5 is not entirely new. However, the view emphasizes the need for domain knowledge in practice where there is a tradeoff between the knowledge requirement and the data requirement in an analytic task, even though such a tradeoff might not be easily quantifiable.

In view of Figure 1.5, it is obvious that the effectiveness of the learning is not determined solely by the machine learning tool. For example, if relevant features are missing in a dataset, then no tool can produce a model completely capturing the underlying answer. Consequently, automation of the entire search process requires automation of the steps in the Analyst Layer as well. This means that automation

requires implementing a way to acquire and model the analyst's domain knowledge and this knowledge acquisition can be very much application dependent (see, e.g. [9][13] [14][15]).

### *1.2.2 The model evaluation step*

In practice, the model evaluation step can be expensive and/or time consuming. For example, in functional verification, the step might involve modifying the testbench according to a learned rule and generating some new tests. Then, these new tests are simulated to decide if the rule is meaningful. In other applications, the evaluation might involve meetings with the design team. In the context of production yield optimization, the meetings could involve engineers from a foundry outside the company. Consequently, each search iteration can be delayed and this bottleneck is outside the machine learning toolbox.

### *1.2.3 The tool requirement*

To speed up the search process, ideally the search desires a machine learning tool that reports not only a learned model but also a quality measure for the model. This quality measure can help decide in the model evaluation step whether or not to go through an expensive evaluation process.

In traditional machine learning, a tool is designed to output an "optimized" model based on a given dataset where the optimization objective depends on the learning algorithm. Then, the quality of the model is evaluated through *cross-validation* separated from the learning algorithm. In cross-validation, there are two datasets: a *training* dataset and a *validation* dataset. The model is learned with the training dataset and its accuracy is calculated by applying the model to the validation dataset. The accuracy seen on the validation dataset is supposed to represent how the model will perform if it is applied on future unseen samples.

While cross-validation is a common practice, the no-free-lunch (NFL) theorem in machine learning [12] warns about its misuse in practice. Unless one can ensure that the validation dataset is somewhat a complete representation for the future unseen samples (which is often not the case in practice), cross-validation may provide little assurance in practice for an application.

More importantly, for the applications mentioned before, in practice cross-validation might not be a viable option to assess the quality of a model. This is due to the limitation on the positive samples and/or on the total samples as discussed above. This means that in Figure 1.5, the quality of a model has to be assessed in a separate model evaluation step which can be expensive. The only possible assurance a learning tool can provide on its output models is that they are "optimized" with respect to some optimization objective. However, whether such an optimization ob-

jective is meaningful with respect to the particular application context can be quite difficult to assess.

In view of Fig. 1.5, ideally, the search desires a machine learning tool that by itself can provide some quality assurance for its output model. This can alleviate the need for cross-validation. This requirement leads to the main question for the machine learning toolbox: What additional quality assurance a learning algorithm can provide?

In summary, there are two areas of concern in an iterative search process: (1) How to provide more quality assurance for the models entering the model evaluation step? (2) How to effectively incorporate the domain knowledge in the iterative search process? In the rest of the chapter, the discussion will focus more on the first question. Then, in section 1.7 we briefly review several recent works [9][13] [14][15] related to the second question.

## 1.3 Assumptions in Machine Learning

To address the model quality concern, we need to understand why cross-validation is needed in the first place, i.e. why a machine learning algorithm does not provide an assurance for its model quality and demands cross-validation to evaluate its model? To facilitate the discussion, Figure 1.6 illustrates a theoretical setup in the context of *supervised learning*.

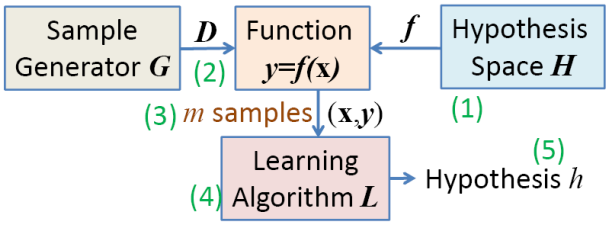


**Fig. 1.6** Five areas to make an assumption in order to enable learning

In this setup, a hypothesis space $H$ is assumed. $H$ is a set of functions (hypotheses) and one of them $f$ is the target function (true answer) to be learned. A sample generator $G$ produces a set of $m$ samples $\mathbf{x}_1, \ldots, \mathbf{x}_m$ according to an unknown but fixed distribution $D$. For each sample $\mathbf{x}_i$, its label $y_i$ is calculated as $f(\mathbf{x}_i)$. Then, the dataset comprising the $m$ pairs $(\mathbf{x}_i, y_i)$, is given to a learning algorithm $L$ to learn. The algorithm $L$ outputs its answer $h$. Ideally, if the answer is correct, we would have $\forall \mathbf{x}$ generated from $G, f(\mathbf{x}) = h(\mathbf{x})$.

In theory, $f$ has to be *learnable* [16][17] in order for a learning algorithm to achieve some sort of learning. To ensure learnability, some assumptions need to be

made in view of the setup. There are five areas to make an assumption, as marked in Figure 1.6.

The first assumption concerns the hypothesis space $H$. It is intuitive that learnability depends on the complexity of $H$, i.e. the more complex the $H$ is, the more difficult the learning is (hence less learnable). If $H$ is finite and enumerable, then its complexity can be measured more easily. For example, if $H$ is the set of all Boolean functions based on $n$ variables, then $H$ contains $2^{2^n}$ distinct functions.

The difficulty is when $H$ is infinite and/or uncountable. In this case, one cannot rely on counting to define its complexity. One theory to measure the complexity of $H$ is based on its ability to fit the data. This concept is called the *capacity* of $H$ which is characterized as the *VC dimension* [18]. The VC dimension (VC-D) also represents the minimum number of samples required to identify a $f$ randomly chosen from $H$. To learn, one needs to make an assumption on the VC-D, for example VC-D should be on the order of $poly(n)$ (polynomial in $n$, the number of features). Otherwise, the number of required samples can be too large for the learning to be practical.

The second assumption concerns the sample generator $G$. The common assumption is that $G$ produces its samples by drawing a sample randomly according to a fixed distribution $D$. Hence, as far as the learning concerns, all future samples are generated according to the same distribution.

The third assumption concerns the number of samples ($m$) available to the learning algorithm. This $m$ has to be at least as large as the VC-D. Then, the fourth assumption concerns the complexity of the learning algorithm. Even though $m$ is sufficiently large, learning the function $f$ can still be computationally hard [17].

The computational hardness can be characterized in terms of the traditional NP-Hardness notion [17] or the hardness to break a cryptography function [19]. For example, learning a 3-term DNF (Disjunctive Normal Form) formula using the DNF representation is hard [17]. In fact, except for a few special cases, learning based on a Boolean functional class is usually hard [17]. Moreover, learning based on a simple neural network is hard [20]. The computational hardness implies that in practice for most of the interesting learning problems, the learning algorithm can only be a heuristic. Consequently, its performance cannot be guaranteed on all problem instances.

The last assumption concerns how the answer $h$ is evaluated. In the discussion of the other assumptions above, we implicitly assume that the "closeness" of $h$ to $f$ is evaluated through an error function $Err()$, for example $Err(h,f) = Prob(h(\mathbf{x}) \neq f(\mathbf{x}))$ for a randomly drawn $\mathbf{x}$. Notice that with such an $Err()$, an acceptable answer does not require $h = f$. As far as the learning concerns, as long as their outputs are the same with a high probability, $h$ is an acceptable answer for $f$. This is because the purpose of the learning is for prediction and the goal is to have a predictor whose accuracy is high enough. However, for many applications in design and test processes, the use of a learning model is not for prediction but for interpretation. For those applications, adopting such an error function can be misleading.

## 1.4 Traditional Machine Learning

When applying a machine learning algorithm, a practitioner is often instructed to pay attention to the issue of model *overfitting*. In practice, one way to observe overfitting is through cross-validation. Let $D_T$ and $D_V$ denote the training and validation datasets. Let $EmErr(h,D)$ be an error function to report an *empirical error rate* by applying a model $h$ onto the dataset $D$. Let the learning error rate be $e_T = EmErr(h,D_T)$ and validation error rate be $e_V = EmErr(h,D_V)$. In learning, a learning algorithm has only $D_T$ to work on. Hence, the algorithm can try to improve on $e_T$ but does not know what the resulting $e_V$ might look like.

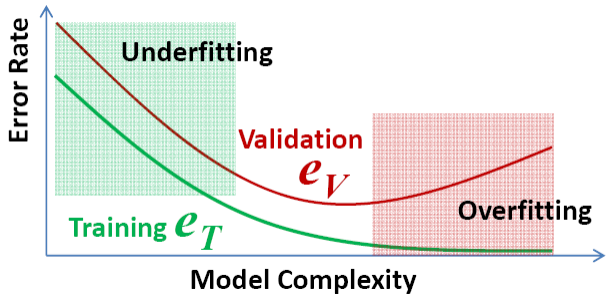


**Fig. 1.7** Underfitting Vs. Overfitting

Overfitting means that the learning algorithm continues to improve on $e_T$, but $e_T$ and $e_V$ deviate from each other and hence, the improvement does not translate to $e_V$. In contrast, *underfitting* means that $e_T$ is high and hence there is still room for improvement. Fig. 1.7 illustrates these concepts where the x-axis can be thought of as a scale to reduce $e_T$ based on employing a more complex model.

Refer back to Figure 1.6. A learning algorithm usually has no knowledge regarding the actual hypothesis space $H$ where the function $f$ is drawn. To learn, the learning algorithm assumes a hypothesis space $H_L$ to begin with. This is usually done by assuming a model representation, for example, such as a particular neural network design. When the assumption of $H_L$ is over-constrained, its capacity is smaller than the capacity of $H$. Then, it is possible that $H_L$ does not contain a hypothesis $h$ that is close enough to $f$. As a result, there is little chance for $Err(h,f)$ to approach zero. This can be considered as another perspective to understand the concept of underfitting.

In practice, to avoid underfitting the learning begins with an assumed $H_L$ that is as less constrained as possible, i.e. with a capacity as large as possible. In practice, this assumption is constrained by the computational resources for the learning. This is because assuming a more complex $H_L$ usually implies more computational overhead in the learning. Also, assuming a more complex $H_L$ means more samples needed to cover the hypothesis space. More importantly, with a $H_L$ whose capac-

ity is larger than the capacity of $H$, obtaining sufficient data samples to achieve a complete coverage on $H_L$ might not be practically possible.

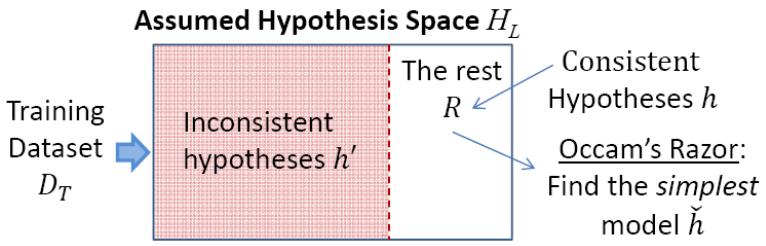


**Fig. 1.8** Occam's Razor in machine learning

For example, Figure 1.8 depicts such a situation. After a learning algorithm checks the hypotheses in a hypothesis space $H_L$ against all samples in the training dataset $D_T$, the space can be divided into two subsets. The first includes all hypotheses $h'$ that are inconsistent with the samples, for example $EmErr(h', D_T) \neq 0$. Then, the rest $R$ (it is called the *version space*) includes all *consistent* hypothesis $h$ where $EmErr(h, D_T) = 0$. In overfitting, $R$ contains two or more distinct answers. Here "distinct" means existing one $\mathbf{x}$ where $\mathbf{x} \notin D_T$ and the two hypotheses $h_1, h_2$ result in two different values, i.e. $h_1(\mathbf{x}) \neq h_2(\mathbf{x})$. In other words, there exists a sample to differentiate $h_1$ and $h_2$ but this sample is not in $D_T$. In practice $R$ can contain a large number of distinct hypotheses after learning on $D_T$, for example due to insufficient samples used in the learning.

### *1.4.1 Occam's Razor*

In machine learning, a common strategy to pick a model in the version space $R$ is based on the Occam's Razor principle, i.e. picking the "simplest" model as the answer. Applying the Occam's Razor principle requires a measure for model simplicity. Suppose this measure is defined. Then picking the model in $R$ becomes an optimization problem, i.e. optimizing the model according to the simplicity measure. Because such an optimization problem can be computationally hard, a heuristic is usually developed to tackle the problem and such a heuristic does not guarantee always finding the optimal model.

In theory, there is also some subtlety to apply the Occam's Razor principle in learning [21]. Furthermore, the definition of a simplicity measure might or might not have a physical meaning in an application context. For all those reasons, an "optimized" model reported by a learning tool might provide little assurance on $e_V$. Consequently, cross-validation is required to evaluate the model even though it is considered as an optimized model by a learning algorithm.

### *1.4.2 Avoid overfitting*

Ideally, one would like to assume a $H_L$ whose capacity is the same as $H$. However, this can be extremely difficult to accomplish in practice. In the context of feature-based analytics discussed before, underfitting can mean a required feature is not included in the initial feature set. Overfitting can mean there are features included where no data are available to tell their relevance. To avoid underfitting, one desires to begin with an initial feature set containing all possible features. However, this strategy can lead to a problem where there is no sufficient data to find the exact answer (when those features are considered together).

From this perspective, the iterative search process in Figure 1.5 can be thought of as the search for the right hypothesis space $H_L$. Because the data might not be sufficient for the search, the Analyst Layer is needed to assist the search based on domain knowledge.

## 1.5 An Adjusted Machine Learning View

As discussed in Section 1.4, a traditional machine learning algorithm is designed to find an optimized model that fits a dataset based on a given hypothesis space assumption. In practice, such an approach may provide little quality assurance on its answer and hence, increases the burden of the model evaluation step in Figure 1.5. In addition, cross-validation might not be a viable option for the applications considered due to limitation on the available samples for the learning.

For feature-based analytics, the essence of the problem is finding the right hypothesis space assumption. From this perspective, it is desirable to design a machine learning tool that can automatically evaluate a hypothesis space assumption before finding a fitting model. With this in mind, an adjusted view for the machine learning algorithm can be stated as the following: To search for a *hypothesis space assumption* where there is exactly one hypothesis that fits all samples in a given dataset.

In other words, the adjusted machine learning view adds an additional constraint for judging the quality of the learning – The resulting hypothesis has to be unique in terms of the assumed hypothesis space.

It is interesting to note that the adjusted machine learning view is compatible with the Occam's Razor principle and with the Structural Risk Minimization (SRM) proposed in [18]. The difference is the addition of the uniqueness requirement. In fact, an early work to justify the Occam's Razor principle in machine learnig [22] already suggested that the simplest model is better because the model is more likely to be unique, i.e. it is harder to find another answer with the same complexity which can fit the data. In a sense, the adjusted machine learning view makes the uniqueness

an explicit requirement for learning, which is already implicitly hinted by adopting the Occam's Razor principle.

### 1.5.1 Search for a hypothesis space assumption

To implement the adjusted machine learning view, we need a way to determine the uniqueness for a given hypothesis space assumption. In addition, we also need a way to define a set of hypothesis space assumptions. To facilitate the search we may desire to order those hypothesis spaces with increasing capacity as $H_1, H_2, \ldots, H_i, H_{i+1}, \ldots$ where $H_p$ is less complex than $H_q$ for all $p < q$. This ordering enables one to evaluate a set of hypothesis spaces incrementally by following the Occam's Razor principle (and the SRM in [18]). Figure 1.9 illustrates the idea.

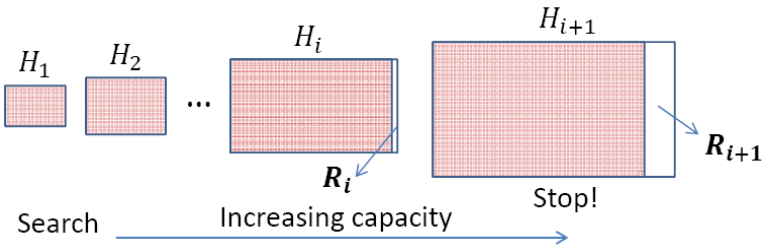


**Fig. 1.9** Search for a hypothesis space assumption

In this figure, a version space $R_i$ represents the set of consistent hypotheses (see Figure 1.8) in $H_i$ based on a given dataset. In the figure, $R_1, \ldots, R_{i-1}$ are all empty, meaning that no hypothesis in those hypothesis spaces can be found to fit all the samples in the dataset (i.e. these hypothesis spaces underfits the dataset). $H_i$ is the first hypothesis space where $R_i$ is not empty. Depending on how stringent the uniqueness requirement is applied, the search may continue into $H_{i+1}$. For example, uniqueness may be defined as $|R| = 1$, i.e. containing exactly one consistent hypotheses. Alternatively, the requirement might be relaxed to be such as $0 < |R| \leq 5$. If the requirement is satisfied by $H_i$, then the consistent hypothesis (or hypotheses) is reported.

A machine learning tool implementing the idea in Figure 1.9 can provide two advantages. First, the output includes a hypothesis space assumption used to obtain the answer(s). This provides additional information for its user to judge the meaningfulness of a reported model. Second, it is possible that the tool results in a situation where no hypothesis space satisfies the uniqueness requirement. In this case, the learning can fail. This failure can immediately triggers an adjustment to the feature set in use or the set of the hypothesis spaces assumed. This means to start another iteration in the search process in Figure 1.5, without involving the expensive model evaluation step.

## 1.6 A SAT-Based Implementation

One major challenge for implementing a learning tool following the idea presented in Figure 1.9 is to obtain an ordered sequence of hypothesis spaces. This ordering should be based on measuring the capacities of the hypothesis spaces. If a hypothesis space contains an infinite number of hypotheses, it is difficult to measure its capacity. In theory the capacity can be measured in terms of its VC dimension [18]. However, in practice the *effective capacity* of a hypothesis space is also limited by the learning algorithm [6], making its estimation quite difficult.

For applications discussed in Section 1.1, however, the number of features is limited. Moreover, the number of values a feature can take can be limited as well. Hence, for those applications we can assume that each feature has only a limited number of possible values. If we use a pseudo feature to represent a particular feature value, then we can further assume that all the features in use are binary, i.e. yes for the occurrence of the feature value and no otherwise. Therefore, from a theoretical perspective the underlying learning problem can be treated as a Boolean learning problem.

### *1.6.1 Boolean learning*

Given $n$ features, the Boolean space contains $2^{2^n}$ Boolean functions. A hypothesis space is a set of Boolean functions in this space. Usually, a hypothesis space is specified with a representation. For example, a $k$-term DNF restricts the functions to those representable with $k$ product terms. Each product term (also called a *monomial*) can have up to $n$ literals (features in positive or negative forms).

For a given Boolean hypothesis space, its capacity can be measured in terms of the number of Boolean functions contained in the space. Hence, assuming that the underlying learning problem is Boolean learning avoids the difficulty for measuring the capacity of an infinite hypothesis space. However, it does not avoid the computational hardness discussed in Section 1.3 before. For example, learning DNF formulas is as hard as solving a random K-SAT problem [24]. For $k$-term DNF, even for $k = 3$ the problem is hard [17] (hard to find a polynomial-time algorithm unless $\mathbf{RP} = \mathbf{NP}$. Note: $\mathbf{P} \subseteq \mathbf{RP} \subseteq \mathbf{NP}$ [23]).

### *1.6.2 Monomial learning*

The simplest case for $k$-term DNF learning is $k = 1$. In this case, the problem is essentially to learn a *monomial*. From the perspective of Computational Learning Theory (CLT), monomials are efficiently learnable [17]. Even though monomial learning is considered as an easy problem in CLT, it can still be a hard problem in view of Figure 1.9. In Fig. 1.6, consider that $D$ is a uniform distribution. Suppose

the true answer $f$ is the monomial $f_1 \cdots f_j$ for a large $j$. It is likely that no positive sample is generated even for a large $m$, i.e. for all samples $\mathbf{x}$ produced by $G$ we have $f(\mathbf{x}) = 0$. Consequently, the output model is simply the constant 0. From the CLT perspective, the constant 0 is a good answer because the error probability for $f(\mathbf{x}) \neq 0$ on a randomly-drawn $\mathbf{x}$ is extremely low. However, in practice a constant 0 most often means the learning has failed.

With the adjusted learning view Figure 1.9, the concept of learnable is not based on an error function $Err()$. Instead, learnable can be viewed as achieving a version space with a very small $|R_i|$ (e.g. $|R_i| = 1$). With this change, monomial learning can be hard when there lacks a positive sample. In fact, if there is only one positive sample and many negative samples, finding the shortest monomial is NP-hard [25].

### *1.6.3 Hypothesis space pruning*

Take monomial learning as an example. The ordered sequence of hypothesis spaces in Figure 1.9 can be defined as $H_1, \ldots, H_n$ where $n$ is the number of features. Each $H_l$ comprises the monomials of the same length $l$ (i.e. with $l$ literals). For example, $H_1$ comprises the monomials of length 1, i.e. $\{f_1, f_1', f_2, f_2', \ldots, f_n, f_n'\}$. $H_2$ comprises the monomials of length 2, which has $2^2\binom{n}{2}$ monomials: For every pair of features $f_i, f_j$, where $i \neq j$, we have $2^2$ monomials $\{f_i f_j, f_i' f_j, f_i f_j', f_i' f_j'\}$. In general, $H_l$ comprises $2^l\binom{n}{l}$ monomials.

For a given $H_l$ and a dataset, our goal is to compute the version space $R_l$. This computation can be based on removing the monomials that are inconsistent with the samples. For example, suppose $n = 3$ and $l = 2$. A negative sample "001" removes the following hypotheses: $f_1' f_2'$, $f_1' f_3$, $f_2' f_3$. A positive sample "100" removes all hypotheses except for: $f_1 f_2'$, $f_1 f_3'$, $f_2' f_3'$. In general, a negative sample removes $\binom{n}{l}$ hypotheses while a positive sample removes all but the $\binom{n}{l}$ hypotheses. Therefore, the pruning power of a positive sample is larger than a negative sample.

The ordered sequence of hypothesis spaces can be generalized to include $k$-term DNF learning for up to a small $k$, for example $k \leq 3$. In this case, there are two variables to define an ordering of the hypothesis spaces, $l$ as the total number of literals in a hypothesis and $k$ as the total number of terms. For example, the hypothesis spaces can be ordered in such a way that $H_{l_1,k_1}$ is before $H_{l_2,k_2}$ in the sequence if $l_1 < l_2$ or $l_1 = l_2 \wedge k_1 < k_2$.

In pruning a hypothesis space $H$, each sample can be seen as a constraint that removes a subset of hypotheses in the space. Suppose our uniqueness requirement is $|R| = 1$, i.e. the version space after the pruning contains exactly one hypothesis. Then the pruning can be formulated as a Boolean Satisfiability (SAT) problem. A satisfiable assignment represents a hypothesis that is consistent with all samples. To check for the uniqueness requirement we need to invoke SAT twice. The first time is to find a satisfiable assignment $A$. Then, $A$ is converted into a constraint to block itself. After this constraint is added to the SAT formula, if the result is unsatisfiable then we know $A$ is the only satisfiable assignment. If not, we know $|R| > 1$.

### 1.6.4 SAT encoding

To convert the pruning problem into a SAT problem, we need three groups of CNF (Conjunctive Normal Form) clauses: (1) clauses to constrain the hypothesis space based on given $n, l, k$, where $n$ is the number of features, $l$ is the number of literals in a hypothesis, and $k$ is the number of terms; (2) clauses to constrain the space based on positive samples (if any), and (3) clauses to constrain the space based on negative samples. Let $m_p$ be the number of positive samples and $m_n$ be the number of negative samples. Then, the encoding method described below results in a CNF formula with $\Theta(nkl + km_p)$ symbols and $\Theta(nkl + nkm_p + km_n)$ clauses.

The key idea for the encoding is that each feature can appear in positive, or in negative, or does not appear at all in a term. Hence, three variables are used to represent these three cases for a feature:

- $X^j_{i,1}$ is True iff the $i$-th feature in the $j$-th term appears in negative form
- $X^j_{i,2}$ is True iff the $i$-th feature in the $j$-th term appears in positive form
- $X^j_{i,3}$ is True iff the $i$-th feature does not appear in the $j$-th term

Since exactly one of the three cases is true, one-hot constraints are required to enforce the requirement:

$$\Pi^k_{j=1}\Pi^n_{i=1}(X^j_{i,1} + X^j_{i,2} + X^j_{i,3})(\neg X^j_{i,1} + \neg X^j_{i,2})$$
$$(\neg X^j_{i,1} + \neg X^j_{i,3})(\neg X^j_{i,2} + \neg X^j_{i,3}).$$

For a given $(l,k)$, we need to constrain the space to those hypotheses containing only $l$ literals. This involves a cardinality constraint.The performance of different encoding methods for a cardinality constraint can be found in [26]. In our implementation, we choose the sequential counter method [27] because its performance is comparable to other encoding methods and it has the unit propagation property[26]. The encoding for the cardinality constraint requires additional $l(nk-1)$ new symbols and $\Theta(nkl)$ clauses. Further detail can be found in [28].

To illustrate the conversion from a positive samples into clauses, consider a positive sample $s = 101$. For a single term to be evaluated as true, feature 1 and feature 3 must not appear in negative form and feature 2 must not appear in positive form. Then, at least one term must be evaluated as true. A naive encoding leads to $n^k$ clauses, which is not feasible. To overcome this challenge, additional $k$ symbols, $A^1, A^2, \ldots, A^k$ are used such that $A^j$ is true if and only if the $j$-th term is evaluated as true. The number of clauses reduces to $(n+1)k+1$. The requirement of at least one term is evaluated as true is encoded by a single clause:

$$(\Sigma^k_{j=1}A^j),$$

and for each $j$, the relation of $A^j$ and $X^j_{i,\delta}$ is maintained by

$$\Pi^n_{i=1}(\neg X^j_{i,2-s[i]} + \neg A^j), \text{ and}$$
$$(\Sigma^n_{i=1}X^j_{i,2-s[i]} + A^j).$$

Similarly, suppose $s = 101$ is a negative sample. For a single term to be evaluated as false, at least one of feature 1 and feature 3 must appear in negative form or feature 2 appear in positive form. Besides, all the terms must be evaluated as false. For each sample, $k$ clauses are required. The overall encoding is

$$\Pi_{j=1}^{k}(\Sigma_{i=1}^{n} X_{i,2-s[i]}^{j}).$$

### *1.6.5 Effect of the uniqueness requirement*

To illustrate the effect of including the uniqueness requirement in learning, in this section the performance of the SAT-based learning tool is compared to a popular decision-tree learning tool, the CART tool from the Python machine learning library [10]. The discussion focuses on why uniqueness can be a desirable property in learning. Detail of our learning tool is described in [28] which is named VeSC-CoL (Version Space Cardinality based Concept Learning) and uses the Lingeling solver [29] for SAT.

In the experiment, we assume the number of features $n = 100$. We further assume the length of the true answer is 6 which can be a $k$-term DNF formula for $k = 1, 2, 3$. In each case, a $k$ is randomly picked and the true answer is also randomly picked from all the $k$-term DNF hypotheses. The dataset contains exactly $k$ positive samples which is randomly generated. Then, negative samples are also randomly generated.

Figure 1.10 shows a comparison result for VeSC-CoL and CART. The experiment includes 20 cases. The x-axis shows the number of negative samples used in the learning up to that particular point. In the experiment, the $k$ positive samples are always used first, before any negative samples are used.

For CART, the figure shows the number of cases the CART tool correctly reports the true answer at each x value. For VeSC-CoL, the figure shows two numbers at each x value. The first is the number of cases where VeSC-CoL reports a unique hypothesis as its answer. This is marked as a red dot. The second is the number of cases where VeSC-CoL correctly finds the true answer. This is marked as a blue dot. Where these two numbers coincide, the figure shows an overlap of red dots circled by blue edge.

For CART, the best scenario is that it finds the true answer in 6 out of the 20 cases. This happens occasionally on some particular x values after 3000. For the range of the x values shown, CART performs slightly better as more samples are provided. However, in addition to being poor, its performance fluctuates quite frequently as more samples are added.

For VeSC-CoL, notice that a unique hypothesis found by the tool does not always guarantee it is the correct answer. However, this happens only when the x value is still relatively small. After about x= 650, a unique hypothesis is always the correct answer. More interestingly from x= 650 and up to about x= 1500, as more samples are added to the learning, the VeSC-CoL result always improves. After x= 1500, VeSC-CoL finds all the 20 true answers and the result does not change with more samples added.

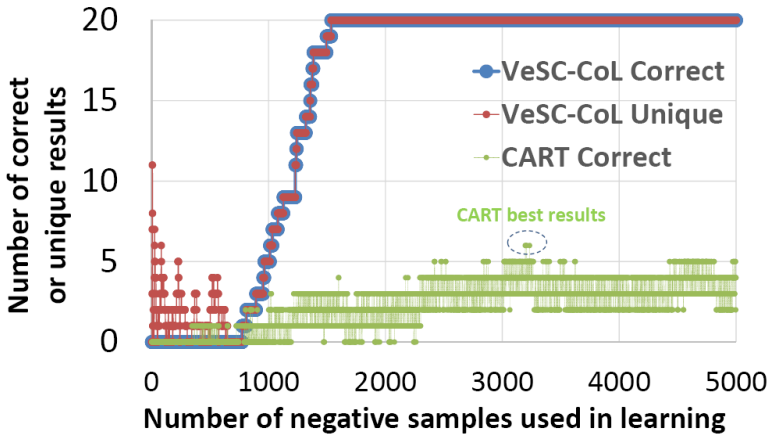


**Fig. 1.10** A comparison between VeSC-CoL and CART

Figure 1.11 presents the result of VeSC-CoL from a different perspective and focuses on x value up to 800. For each of the y label from 1 to 20, the figure marks the x values where VeSC-CoL finds a unique hypothesis but it is not the true answer. We can call each range of such x values a *mistake window*. Figure 1.11 shows where such mistake windows occur as the number of samples increases.

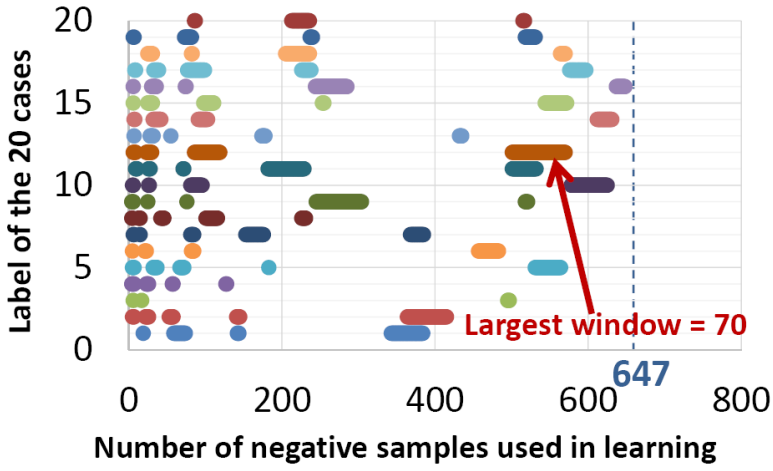


**Fig. 1.11** The mistake windows in VeSC-CoL learning when the x value is small

It is interesting to observe that most of the mistake windows occur when the x value is less than 200. The largest window size is 70. Most importantly, a mistake

window occurs only for x $\leq$ 647. After more than 647 samples are used in the learning, no mistake window occurs (also see Figure 1.10 above).

To see what a mistake window means, refer back to the hypothesis space search process depicted in Figure 1.9 before. Suppose $H_i$ is the hypothesis space that contains the true answer. When the samples used in the learning are not sufficient, the version space $R_i$ can still include many unfiltered hypotheses. However, it is possible that the samples are enough to filter out most of the hypotheses in a simpler hypothesis space, say $H_2$. By chance, this filtering might result in one hypothesis left in $H_2$. Then, because of its uniqueness, VeSC-CoL would report the hypothesis as its answer. However, as more samples are added to the learning, this hypothesis will be filtered out and the wrong answer is removed. In other words, the largest window size 70 shown in Figure 1.11 means that such a mistake made by VeSC-CoL is corrected after 70 additional samples are used. From this perspective, Figure 1.11 shows that in all cases when VeSC-CoL makes a mistake, the mistake is corrected after up to 70 new samples added in the learning.

Note that each mistake window shown in Figure 1.11 corresponds to a hypothesis space that is simpler to the hypothesis space containing the true answer. Because the true answer is in a hypothesis space with $l = 6$ and $k$ randomly selected between 1 and 3, there are many hypothesis spaces before the true answer as the hypothesis spaces are arranged according to the ordering discussed in Section 1.6.3.

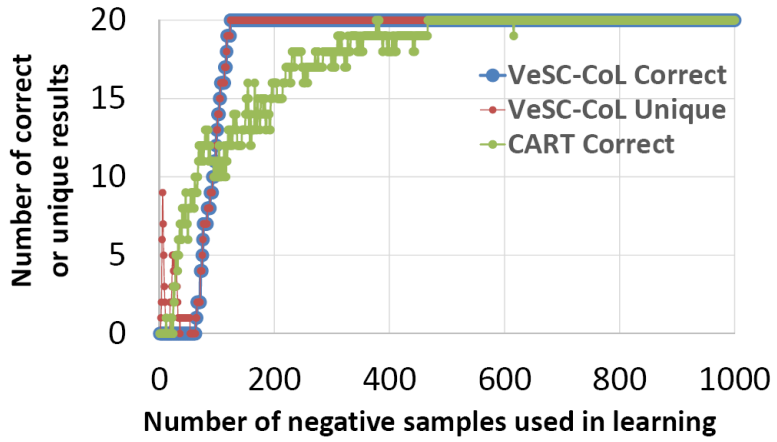


**Fig. 1.12** Result by redoing the experiment with $l = 3$ and $k = 1, 2$

Figure 1.12 shows the result by redoing the experiment with $l = 3$ and $k$ randomly selected between 1 and 2. In other words, the result shows what happens if we make the true answer easier to learn. For VeSC-CoL, its performance is similar to that shown in Figure 1.10 before. After x $>$ 124, VeSC-CoL finds the true answer in all 20 cases. Recall that this number is about 1500 in Figure 1.10 before.

The performance of CART is substantially better than that shown in Figure 1.10. It is interesting to observe that in some cases (only occur when x$<$ 90), CART can actually perform better than VeSC-CoL, i.e. finding true answers for more cases. However, CART result can still fluctuate as more samples are used. But eventually (after about x $>$ 500), CART can also finds all true answers.

The two results presented above can be summarized as the following:

- Finding a unique answer in a hypothesis space does not guarantee it is the correct answer. However, if there are enough samples to filter out all other hypotheses in the hypothesis space containing the true answer, the true answer is guaranteed to be found.
- After some initial samples, the performance of VeSC-CoL becomes *consistent*, i.e. more samples always leading to a better result. This is not the case for CART.
- If the true answer is easy to learn, CART can be better than VeSC-CoL, i.e. can find the true answer with fewer samples. However, the difference in the number of samples required to learn between CART and VeSC-CoL is not that significant. In contrast, if the true answer is hard to learn, VeSC-CoL can significantly out-perform CART.
- For a hard-to-learn answer, VeSC-CoL requires much less samples to learn than CART. However, because VeSC-CoL involves a SAT solver, it is computationally more expensive. In a sense, VeSC-CoL enables its user to trade computational cost for sample requirement. Hence, if samples are limited, VeSC-CoL can be a useful alternative for learning.

To conclude this section, Figure 1.13 shows another result with the same setting as that for generating Figure 1.10 by adding 100 more cases. As seen, the performance of VeSC-CoL in Figure 1.13 is similar to that shown in Figure 1.10. The performance of CART is also comparable, at x$=$ 5000 CART finds the true answer for 22 cases. The difference between VeSC-CoL and CART remains significant.

Figure 1.14 then shows what happens to the CART's performance if the number of negative samples is increased up to 50000. Close to the right end in the figure, CART correctly finds the true answer for 45 cases. For additional 8 cases, CART finds the true answer occasionally at some particular x values but the result is not stable. As a result, the number of correctly found cases fluctuates between 47 and 53 on the right side of the figure.

Table 1.1 then shows what types of the cases CART can find the true answer and what types of the cases CART cannot. Recall that the length of each true answer is fixed at 6. A true answer can be a 1-term, 2-term, or 3-term DNF formula. If it is a 1-term DNF, it is a length-6 monomial. If it is a 2-term DNF, the lengths of the two terms are represented as $(l_1, l_2)$ where $l_1 + l_2 = 6$. Similarly, for a 3-term DNF, the lengths are represented as $(l_1, l_2, l_3)$ where $l_1 + l_2 + l_3 = 6$.

Observe from Table 1.1 that if the true answer is a 1-term monomial, CART can find the true answer for all of them. If it is a 2-term DNF with lengths (1,5), CART can also find the true answer. For those (1,1,4) cases, CART's result fluctuates

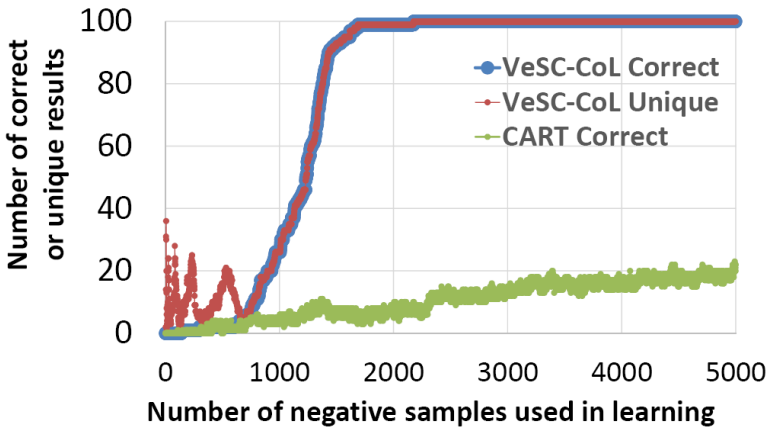


**Fig. 1.13** Result by redoing the experiment with 100 cases

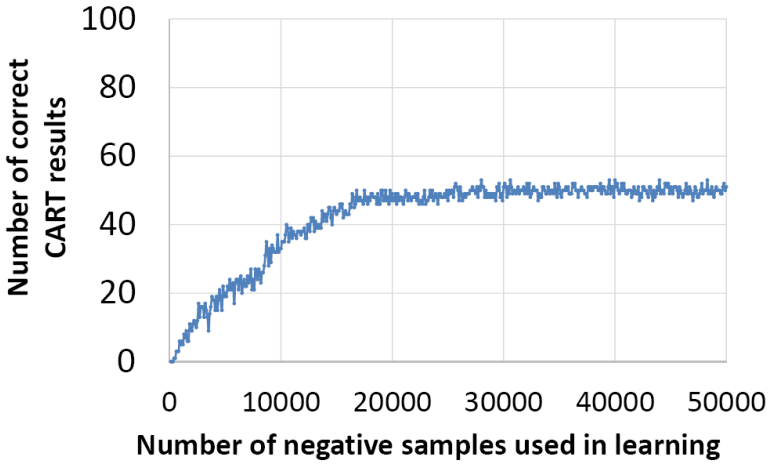


**Fig. 1.14** CART result by adding up to 50000 negative samples in learning

between 2 and 8. For other cases, CART does not find the true answer at any x value even with up to 50000 negative samples in use.

Suppose we only consider those 53 cases CART can find the true answer at least once. Figure 1.14 shows that it still requires about 20000 negative samples for CART to find all those answers. This is compared to the sample requirement for VeSC-CoL in Figure 1.13, where after about 1700 samples VeSC-CoL correctly finds all 100 true answers.

From the results presented in this section, observe that the performance of VeSC-CoL is quite consistent. In contrast, the performance of CART is not. In Figures 1.10 1.13, and 1.14 CART's performance does not look satisfactory. In Figure 1.12, CART seems to be doing fine. This reveals that the search strategy of Figures 1.9

**Table 1.1** Detail of the CART result for the 100 cases in Figure 1.14

| Formula | Lengths | # cases | # Correctly Found |
|---|---|---|---|
| 1-term | 6 | 35 | 35 |
| 2-term | (1,5) | 10 | 10 |
| 2-term | (2,4) | 10 | 0 |
| 2-term | (3,3) | 15 | 0 |
| 3-term | (1,1,4) | 10 | 2 to 8* |
| 3-term | (1,2,3) | 10 | 0 |
| 3-term | (2,2,2) | 10 | 0 |

*This number fluctuates between 2 and 8

with the uniqueness requirement can result in a learning process more consistent, as comparing to a traditional learning method such as CART whose performance can be largely affected by how easy or difficult to learn the true answer.

## 1.7 Incorporating Domain Knowledge

As discussed in Section 1.2.1, the Analyst Layer in Figure 1.5 is driven mostly by domain knowledge. Hence, to automate the entirety of the iterative search process, one has to consider automation of the steps in the Analyst Layer. The Analyst Layer essentially comprises two major components, the dataset preparation component including the sample selection, feature selection, and dataset construction, and the model (or result) evaluation component.

For example, in the context of functional verification the dataset preparation involves two major tasks: generation of the tests to run the simulation and the selection of signals to encode a simulation trace. The data quality, in this case the quality of the simulation traces largely depends on the tests used in the simulation. To automate the test generation, the work in [14] presents a *constrained process discovery* approach that learns a test generation model based on example tests (e.g. C programs) written by a verification engineer. The test generation model can then be executed to produce tests automatically. For signal selection, often a verification engineer relies on reading the specification document to select the important and relevant signals. To automate this task, the work in [15] develops a text-mining approach to extract signals from a specification document.

In the context of production yield optimization, the dataset preparation involves making several choices to decide what data to analyze and what type of analytic to run [8]. In practice, these choices are made by the analyst. The work in [9] presents a way to learn how an analyst makes such choices in sequence for performing a yield data analytic task. The learning result is captured in a *process model* that looks like a flowchart where each node in the model is a software script. The model can then be executed automatically as if the analyst would perform the analytics for resolving a yield issue. For automating the result evaluation component, recently the work in [13] presents a learning approach to construct a plot recognizer based on example

plots instructed by an analyst. Such a plot recognizer can be used to automatically recognize the meaningfulness of an analytic result when it is presented as a plot. The implementation is based on the recently-proposed Generative Adversarial Networks (GAN) learning approach [30][31].

In general, where to apply learning and what type of learning to apply for automating the Analyst Layer is largely application dependent. For example, in the context of functional verification, learning is applied to automate two different tasks in the dataset preparation and the learning approaches involved are fundamentally different, i.e. constrained process discovery [14] vs. text mining [15]. In the context of production yield optimization, learning is applied to automate both the dataset preparation and result evaluation, and the learning approaches are also different, i.e. process mining [9] vs. GAN [13]. These examples illustrate the diversity of the problem for automating the Analyst Layer. Overall, whether the Analyst Layer for a given application can be fully automated remains an open question to be explored with future research.

## 1.8 Conclusion

In this chapter, the discussion focuses on a particular type of learning encountered in some design and test applications where the data samples are limited, which is given the name *feature-based analytics*. Because of the data limitation, in practice it is more intuitive to view feature-based analytics in terms of an iterative feature search process as depicted in Figure 1.5. The effectiveness of this search depends on the steps conducted in the Analyst Layer as well as on the machine learning tool in use. For the machine learning tool, we explain the challenges to adopt a traditional machine learning problem formulation view. Instead, an adjusted machine learning view is presented, and illustrated in Figure 1.9. In the adjusted view, uniqueness of an answer is included as an additional requirement for learning and the focus of the learning is shifted from finding a model to finding a hypothesis space. A SAT-based approach to realize this adjusted learning view in the context of learning a $k$-term DNF formula is presented and its benefits are illustrated with several experiment results. Finally, for automating the tasks in the Analyst Layer, several recent works are briefly discussed as examples to illustrate the diversity of the problem. More future research is required in order to assess if the Analyst Layer can be fully automated and such automation can also be very much application dependent.

**Acknowledgements** This work is supported in part by National Science Foundation under grant No. 1618118 and in part by Semiconductor Research Corporation with project 2016-CT-2706.

The author would also like to thank his doctoral student Kuo-Kai Hsieh especially for his help on Section 1.6.4 and Section 1.6.5.

This chapter is an extension from author's prior work "Machine Learning for Feature-Based Analytics," in Proceeding 2018 International Symposium on Physical Design, pp. 74-81 ©2018 Association for Computing Machinery, Inc. http://doi.acm.org/10.1145/3177540.3177555. Reprinted by permission.

## References


[1] Li-C. Wang, "Experience of Data Analytics in EDA and Test - Principles, Promises, and Challenges," *IEEE Transactions on CAD*, 36 (6), 2017, pp. 885-898.

[2] Wen Chen, Li-C. Wang, and Jayanta Bhadra, "Simulation Knowledge Extraction and Reuse in Constrained Random Processor Verification," *ACM/IEEE Design Automation Conference*, 2013.

[3] Li-C. Wang, "Data Mining in Functional Test Content Optimization," *ACM/IEEE Asian South Pacific Design Automation Conference*, 2015

[4] Gagi Drmanac, Frank Liu, and Li-C. Wang, "Predicting Variability in Nanoscale Lithography Processes," *ACM/IEEE Design Automation Conference*, 2009.

[5] Janine Chen, Brendon Bolin, Li-C. Wang, Jing Zeng, Dragoljub (Gagi) Drmanac, and Michael Mateja, "Mining AC Delay Measurements for Understanding Speed-limiting Paths," *IEEE International Test Conference*, 2010.

[6] Ian Goodfellow, Yoshua Benjio, and Aaron Courville, *Deep Learning*, The MIT Press, 2016.

[7] Janine Chen, Li-C. Wang, Po-Hsien Chang, Jing Zeng, Stanly Yu, and Michael Metaja, "Data learning techniques and methodology for Fmax prediction," *IEEE International Test Conference*, 2009.

[8] Jeff Tikkanen, Sebastian Siatkowski, Nik Sumikawa, Li-C. Wang and Magdy S. Abadir, "Yield Optimization Using Advanced Statistical Correlation Methods," *IEEE International Test Conference*, 2014.

[9] S. Siatkowski, Li-C. Wang, N. Sumikawa, L. Winemberg, "Learning the Process for Correlation Analysis," *IEEE VLSI Test Symposium*, 2017

[10] Pedregosa et al., "Scikit-learn: Machine Learning in Python," *JMLR*, 2010, pp. 2825-2830.

[11] Nicholas Callegari, Dragoljub (Gagi) Drmanac, Li-C. Wang, Magdy S. Abadir, "Classification rule learning using subgroup discovery of cross-domain attributes responsible for design-silicon mismatch," *ACM/IEEE Design Automation Conference*, 2010, pp. 374–379.

[12] David H. Wolpert, "The Lack of A Priori Distinctions between Learning Algorithms," *Neural Compt.*, 8 (7), pp. 1341-1390.

[13] M. Nero, J. Shan, Li-C. Wang, N. Sumikawa, "Discovering Interesting Plots in Production Yield Data Analytics," *IEEE International Test Conference*, 2018.

[14] K. Hsieh, Li-C. Wang, W. Chen, J. Bhadra, "Learning To Produce Direct Tests for Security Verification Using Constrained Process Discovery," *Design Automation Conference*, June 2017.

[15] K-K. Hsieh, S. Siatkowski, Li-C. Wang, Wen Chen, and Jayanta Bhadra, "Feature Extraction from Design Documents to Enable Rule Learning for Improving Assertion Coverage," *ACM/IEEE Asia South Pacific Design Automation Conference*, 2017.

[16] L. G. Valiant, "A theory of learnable," *Communications of ACM*, 27, 11, 1984, pp. 1134-1142.
[17] Machael J. Kearns and Umesh Vazirani, *An Introduction to Computational Learning Theory*, The MIT Press, 1994.
[18] Vladimir Vapnik, *The Nature of Statistical Learning Theory,* Springer, 2000.
[19] Machael J. Kearns and Umesh Vazirani, "Cryptographic limitations on learning Boolean formulae and finite automata," *Journal of ACM*, 14, 1, 1994, pp. 67-95.
[20] Amit Daniely, Nati Linial, and Shai Shaleve-Shwartz, "From average case complexity to improper learning complexity," *ACM Symposium on Theory of Computing,* 2014, pp. 441-448.
[21] David H. Wolpert, "The Relationship Between Occams Razor and Convergent Guessing," *Complex System*, 4, 1990, pp. 319-368.
[22] J. Pearl, "On The Connection Between The Complexity and Credibility of Inferred Models," *International Journal of General Systems*, Vol 4, pp. 255-264, 1978.
[23] Rajeev Motwani and Prabhakar Raghavani, *Randomized Algorithms*, Cambridge University Press, 1995.
[24] Amit Daniely and Shai Shalev-Shwartz, "Complexity Theoretic Limitations on Learning DNFs," *JMLP*, 49, 2016, pp. 1-16.
[25] David Haussler, "Quantifying Inductive Bias: AI Learning Algorithms and Valiants Learning Framework," *Artificial Intelligence*, 36, 1998, pp. 177-221.
[26] Yael Ben-Haim, Alexander Ivrii, Oded Margalit, and Arie Matsliah, "Perfect hashing and CNF encodings of cardinality constraints," *International Conference on Theory and Applications of Satisfiability Testing,* Springer, 2012, pp. 397-409.
[27] Carsten Sinz, "Towards an optimal CNF encoding of boolean cardinality constraints," *International Conference on Principles and Practice of Constraint Programming,* Springer, 2005, 827831.
[28] K. Hsieh and Li-C. Wang, "A Concept Learning Tool Based On Calculating Version Space Cardinality," `arXiv:1803.08625v1`, Mar 23, 2018.
[29] Armin Biere, "Lingeling, Plingeling and Treengeling entering the SAT competition," *Proceedings of SAT Competition* 2013.
[30] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and J. Bengio, "Generative adversarial networks," `arXiv:1406.2661`, 2014.
[31] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," `arXiv:1511.06434v2`, 2016.