Computational Methods of Feature Selection

CRC PRESS
Boca Raton London New York Washington, D.C.

Contents

0-8493-0052-5/00/$0.00+$.50

List of Tables

0-8493-0052-5/00/$0.00+$.50

List of Figures

0-8493-0052-5/00/$0.00+$.50

Part I

Some Part

Chapter 1

Randomized Feature Selection

David J. Stracuzzi
Arizona State University

1.1 Introduction

Randomization is an algorithmic technique that has been used to produce provably efficient algorithms for a wide variety of problems. For many applications, randomized algorithms are either the simplest or the fastest algorithms available, and sometimes both [16]. This chapter provides an overview of randomization techniques as applied to feature selection. The goal of this chapter is to provide the reader with sufficient background on the topic to stimulate both new applications of existing randomized feature selection methods, and research into new algorithms. Motwani and Raghavan [16] provide a more broad and widely applicable introduction to randomized algorithms.

Learning algorithms must often make choices during execution. Randomization is useful when there are many ways available in which to proceed, but determining a guaranteed good way is difficult. Randomization can also lead to efficient algorithms when the benefits of good choices outweigh the costs of bad choices, or when good choices occur more frequently than bad choices. In the context of feature selection, randomized methods tend to be useful when the space of possible feature subsets is prohibitively large. Likewise, randomization is often called for when deterministic feature selection algorithms are prone to becoming trapped in local optima. In these cases, the ability of randomization to sample the feature subset space is of particular value.

In the next section, we discuss two types of randomization that may be applied to a given problem. We then provide an overview of three complexity classes used in the analysis of randomized algorithms. Following this brief theoretical introduction, we discuss explicit methods for applying randomiza-

0-8493-0052-5/00/$0.00+$.50

tion to feature selection problems, and provide examples. Finally, the chapter concludes with a discussion of several advanced issues in randomization, and a summary of key points related to the topic.

1.2 Types of Randomization

Randomized algorithms can be divided into two broad classes. *Las Vegas* algorithms always output a correct answer, but may require a long time to execute with small probability. One example of a Las Vegas algorithm is the randomized quicksort algorithm (see Cormen, Lieserson and Rivest [4], for example). Randomized quicksort selects a pivot point at random, but always produces a correctly sorted output. The goal of randomization is to avoid degenerate inputs, such as a pre-sorted sequence, which produce the worst-case $O(n^2)$ runtime of the deterministic (pivot point always the same) quicksort algorithm. The effect is that randomized quicksort achieves the *expected* runtime of $O(n \log n)$ with high probability, regardless of input.

Monte Carlo algorithms may output an incorrect answer with small probability, but always complete execution quickly. As an example of a Monte Carlo algorithm, consider the following method for computing the value of π, borrowed from Krauth [11]. Draw a circle inside a square such that the sides of the square are tangent to the circle. Next, toss pebbles (or coins) randomly in the direction of the square. The ratio of pebbles inside the circle to those inside the entire square should be approximately $\frac{\pi}{4}$. Pebbles that land outside the square are ignored.

Notice that the longer the algorithm runs (more pebbles tossed) the more accurate the solution. This is a common, but not required, property of randomized algorithms. Algorithms that generate initial solutions quickly, and then improve them over time are also known as *anytime algorithms* [22]. Anytime algorithms provide a mechanism for trading solution quality against computation time. This approach is particularly relevant to tasks, such as feature selection, in which computing the optimal solution is infeasible.

Some randomized algorithms are neither guaranteed to execute efficiently, nor to produce a correct output. Such algorithms are typically also labeled as Monte Carlo. The type of randomization used for a given problem depends on the nature and needs of the problem. However, note that a Las Vegas algorithm may be converted into a Monte Carlo algorithm by having it output a random (possibly incorrect) answer whenever the algorithm requires more than a specified amount of time to complete. Similarly, a Monte Carlo algorithm may be converted into a Las Vegas algorithm by executing the algorithm repeatedly with independent random choices. This assumes that the solutions produced by the Monte Carlo algorithm can be verified.

1.3 Randomized Complexity Classes

The probabilistic behavior that gives randomized algorithms their power, also makes them difficult to analyze. In this section, we provide a brief introduction to three complexity classes of practical importance for randomized algorithms. Papadimitriou [18] provides a rigorous and detailed discussion of these and other randomized complexity classes. For simplicity, we focus on decision algorithms, or those that output "yes" and "no" answers, for the remainder of this section.

Randomized algorithms are related to nondeterministic algorithms. Nondeterministic algorithms choose, at each step, among zero or more possible next steps, with no specification of which choice should be taken. Contrast this to deterministic algorithms which have exactly one next step available at each step of the algorithm. Note the difference between nondeterministic choices and conditional control structures, such as *if ... then* statements, which are fully determined by the input to the algorithm. A nondeterministic algorithm accepts its input if there exists some sequence of choices that result in a "yes" answer. The well-known class $\mathcal{NP}$ therefore includes languages accepted by nondeterministic algorithms in a polynomial number of steps, while class $\mathcal{P}$ does the same for languages accepted by deterministic algorithms.

Randomized algorithms differ from nondeterministic algorithms in that they accept inputs probabilistically rather than existentially. The randomized complexity classes therefore define probabilistic guarantees that an algorithm must meet. For example, consider the class $\mathcal{RP}$, for *randomized polynomial time*. $\mathcal{RP}$ encompasses algorithms that accept good inputs (members of the underlying language) with non-trivial probability, always reject bad inputs (non-members of the underlying language), and always execute in polynomial time. More formally, a language $L \in \mathcal{RP}$ if some randomized algorithm R accepts string $s \in L$ with probability $\frac{1}{\epsilon}$ for any ϵ that is polynomial in $|s|$, rejects $s' \notin L$ with probability 1, and requires a polynomial number of steps in $|s|$.

Notice that the definition of $\mathcal{RP}$ corresponds to the set of Monte Carlo algorithms that can make mistakes only if the input string is a member of the target language. The complement of the this class, co-$\mathcal{RP}$, then corresponds to the set of algorithms that can make mistakes only if the input string is *not* a member of the target language. Furthermore, the intersection these two classes, $\mathcal{RP} \cap$ co-$\mathcal{RP}$, corresponds to the set of Las Vegas algorithms that execute in worst-case polynomial time.

To see why, first note that each problem in the intersection has two Monte Carlo algorithms. One algorithm never outputs a false positive, while the other never outputs a false negative. By conducting many repeated and independent executions of both algorithms, we are guaranteed to eventually arrive at the correct output. (Recall that Las Vegas algorithms always output the correct answer, but may take a long time to do so.) This intersection is

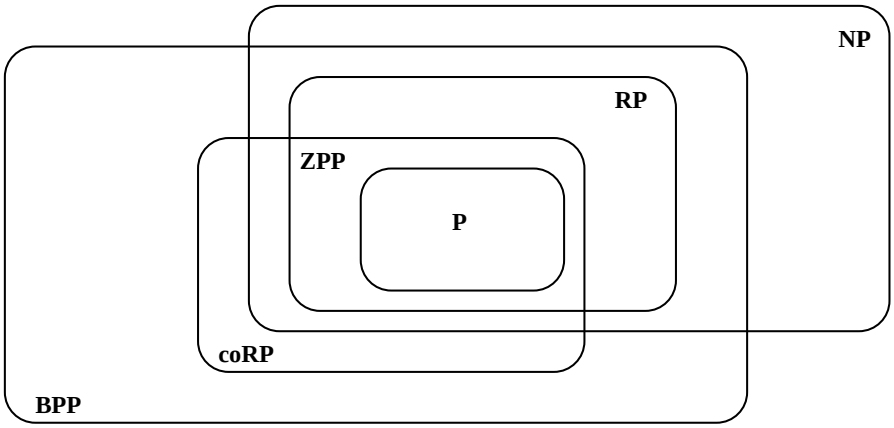

FIGURE 1.1: Illustration of the randomized complexity classes in relation to each other and the deterministic classes P and NP.

also known as the class $\mathcal{ZPP}$, for *polynomial randomized algorithms with zero probability of error*.

In practice we can use algorithms in $\mathcal{RP}$ to construct Monte Carlo algorithms that produce the correct output with high probability simply by running them polynomially many times. If any execution accepts the input, then we return "yes". Since algorithms in $\mathcal{RP}$ never produce false positive results, we can guarantee that the probability of an false negative becomes small. Here, that probability is $(1 - \frac{1}{\epsilon})^k$ for k executions of the algorithm.

The third and largest complexity class of practical importance is $\mathcal{BPP}$, for *polynomial time algorithms with bounded probability of error*. Unlike $\mathcal{RP}$ and $\mathcal{ZPP}$, $\mathcal{BPP}$ allows a randomized algorithm to commit both false positive and false negative errors. This class encompasses algorithms that accept good inputs a majority of the time, and reject bad inputs a majority of the time. More formally, a language $L \in \mathcal{BPP}$ if some randomized algorithm R accepts $s \in L$ with probability $\frac{1}{2} + \frac{1}{\epsilon}$ and accepts $s \notin L$ with probability $\frac{1}{2} - \frac{1}{\epsilon}$ for any ϵ polynomial in $|s|$. Like $\mathcal{RP}$ and $\mathcal{ZPP}$, we can create an algorithm that produces the correct result with high probability simply by executing repeatedly an algorithm that meets the stated minimums.

Figure 1.1 illustrates the relationships among the randomized classes, and shows how the randomized classes related to the deterministic classes P and NP. Note that the figure assumes that $P \neq NP$, which is an open problem. If this assumption turns out to be false, then the complexity classes will collapse into one or just a few classes.

Finally, note that the randomized complexity classes are semantic as opposed to syntactic classes such as $\mathcal{P}$ and $\mathcal{NP}$. Semantic class membership depends on the *meaning* of a specific algorithm instead of the *format* of the

algorithm. For example, we can determine whether an algorithm is a member of class $\mathcal{P}$ by counting the number of times the input is processed. Conversely, we must consider the probability that a given input is *accepted* to determine membership in the class $\mathcal{RP}$. Thus, there is no simple way to check whether a given randomized algorithm fits into a given randomized complexity class. There can be no complete problems for such classes [18].

1.4 Applying Randomization to Feature Selection

A critical step in constructing a randomized algorithm is to decide which aspect of the target problem to randomize. In some cases there may be only one clear option. For example, in the deterministic quicksort algorithm, the pivot is typically chosen arbitrarily as the first element of the current array. However, any fixed choice of pivot would work equally well, so randomizing the selection in an effort to protect against degenerate inputs is successful. Other problems may offer several candidates for randomization.

We formulate the specific feature selection problem considered here as follows. Given a set of supervised training examples described by a set of input features or variables $\mathbf{x}$ and a target concept or function y, produce a subset of the original input variables that predict best the target concept or function when combined into a hypothesis by a learning algorithm. The term "predict best" may be defined in a variety of ways, depending on the specific application. In this context, there are at least two possible sources of randomization.

The first source is the set of input variables. A feature selection algorithm may choose at random which variables to include in a subset. The resulting algorithm searches for the best variable subset by sampling the space of possible subsets. This approach to randomization carries an important advantage. As compared to the popular greedy stepwise search algorithms [1, 8], which add or remove a single variable at a time, randomization protects against local minima. A broad sampling of subsets is unlikely to concentrate effort on any one portion of the search space. Conversely, if many subsets have equally high quality, then a randomized approach will also tend to find a solution quickly.

Randomizing over the set of variables is less likely to be effective if one or a few of the variable subsets is much better than all of the others. The probability of selecting one particular subset at random out of all possible subsets is simply too small. A second issue with this type of randomization is that there is no clear choice of when to stop sampling. A parameter must be set arbitrarily within the algorithm, or the algorithm can be run until the available computation time expires (as an anytime algorithm).

The second possible source of randomization is the set of training examples, often known as the prototype selection problem. If the number of available

examples is very large, an algorithm can select at random which examples to include in a given subset evaluation. The resulting algorithm may conduct a traditional deterministic search through the space of feature subsets, but evaluates those subsets based on a random sample of data. This option is particularly useful when the number of examples available is intractably large, or the available computation time is short.

Notice that as a side-effect, randomization reduces the confidence with which the feature selection algorithm produces results. By sampling only a small portion of the space of variable subsets, we lose confidence that the algorithm's final output is actually the best possible subset. Likewise, when we sample the set of available training data, we lose confidence in the accuracy of our evaluation of a given feature subset. Such effects are of particular concern for algorithms that randomize on both the set of input variables and the set of examples. The approach offers the possibility of combining the advantages of both randomization methods, but also reduces confidence in two ways. Concerns about confidence must be balanced carefully against any reductions in computation.

1.5 The Role of Heuristics

A fundamental goal of computer science is to find correct or optimal problem solutions using a minimum of computation. For many problems, no known algorithm can produce such a solution efficiently. Heuristics are therefore used to relax one or both of these demands on optimality and efficiency.

Randomization itself is a problem solving heuristic. A randomized algorithm may trade optimality for efficiency by searching only a sampled portion of the state space, instead of the entire state space. In many cases there is no guarantee that the best possible solution will be found, but often a relatively good solution is found with an acceptable amount of computation.

Many algorithms employ multiple heuristics. One type of heuristic appropriate to a randomized algorithm is a sampling bias. In the context of feature selection, an algorithm that always samples uniformly from the entire space of feature subsets to obtain its next candidate solution uses randomization as its only heuristic. However, algorithms that bias their samples, for example by sampling only in the neighborhood of the current best solution, employ a second heuristic in conjunction with randomization.

A variety of sampling biases are possible for feature and prototype selection algorithms. We illustrate several examples in the following section. However, not all sampling biases are appropriate to all selection problems. A sampling bias that quickly focuses the search on a small set of features may not be appropriate if there are several disjoints feature sets capable of producing good

learner performance. Likewise, an approach that samples the space broadly throughout the search may not be appropriate if the number of features is large, but few are relevant. As noted above randomization may not be a good choice of heuristic if there is some reason to believe that only a very small number of feature subsets produce desirable results, while all other subsets produce undesirable results. In this case, random sampling is unlikely to uncover the solution efficiently.

Successful application of a randomized (or deterministic) selection algorithm requires some understanding of the underlying feature space. The heuristics and sampling biases used must be appropriate to the given task. Viewed oppositely, successful application of a randomized algorithm *implies* that the underlying feature space exhibits particular characteristics, and these characteristics depend on the specific heuristics used to solve the problem.

1.6 Examples of Randomized Selection Algorithms

We now consider specific examples of randomized feature and prototype selection algorithms. The goal is to illustrate ways in which randomization can be applied to the feature selection problem. We consider both Las Vegas and Monte Carlo methods, a variety of performance guarantees, along with the strengths and weaknesses of each approach. The algorithms discussed here also illustrate a variety of heuristics and sampling biases. As is often the case, no one algorithm uniformly dominates another. The goal of this section is to familiarize readers with existing methods for randomized selection, and to provide the background necessary to make informed choices.

1.6.1 A Simple Las Vegas Approach

The key characteristic of a Las Vegas algorithm is that it must eventually produce the correct solution. In the case of feature selection, this means that the algorithm must produce a minimal subset of features that optimizes some criteria, such as classification accuracy. The Las Vegas Filter (LVF) algorithm discussed by Liu and Setino [12] achieves this goal, albeit under specific conditions.

LVF searches for a minimal subset of features to describe a given set of supervised training examples $\mathbf{X} =< \mathbf{x}_1, y_1 >, \ldots, < \mathbf{x}_M, y_M >$, where $|\mathbf{x}_i| = N$. The subsets are selected uniformly at random with respect to the set of all possible subsets. They are then evaluated according to an inconsistency criterion, which tests the extent to which the reduced-dimension data can still separate the class labels. If the newly selected subset is both smaller in size and has an equal or lesser inconsistency rate, then the subset is retained. LVF

Given:

Examples $\mathbf{X} =< \mathbf{x}_1, y_1 >, \ldots, < \mathbf{x}_M, y_M >$
Maximum allowable inconsistancy γ
Number of attributes N
Number of iterations t_{max}

Algorithm:

$S_{best} \leftarrow$ all N attributes
$c_{best} \leftarrow N$
for $i \leftarrow 1$ **to** t_{max} **do**
    $c \leftarrow$ random number between 0 and c_{best}
    $S \leftarrow$ random selection of c features to include
    if $Inconsistancy(S, \mathbf{X}) \leq \gamma$ **then**
        $S_{best} \leftarrow S$
        $c_{best} \leftarrow c$
return(S_{best})

FIGURE 1.2: The Las Vegas Filter algorithm [12].

performs this simple sampling procedure repeatedly, stopping after a predetermined number of iterations, t_{max}. Figure 1.2 shows the LVF algorithm.

There are two important caveats to the LVF algorithm. First, the algorithm can only be labeled as a Las Vegas algorithm if it is allowed to run sufficiently long to find the optimal solution. For training data described by N input features, we expect to need approximately 2^N iterations. In the case where $t_{max} \ll 2^N$, the algorithm should be considered Monte Carlo. Notice that the Monte Carlo version of the algorithm may be used in an anytime format by returning the current best feature subset at any point during execution.

The second caveat to LVF regards the allowable inconsistency rate, γ. This parameter controls the trade-off between the size of the returned feature subset, and the ability of that subset to distinguish among examples. If we set γ equal to the inconsistency rate of the full data set $\mathbf{X}_{()}$, then LVF is guaranteed to find the optimal solution under the conditions described above for t_{max}. However, a larger inconsistency rate allows LVF to reach smaller feature subsets more quickly. The algorithm then effectively becomes a greedy local search, and is susceptible to local minima. LVF ignores any subset that is selected with size larger than the current best. If a larger subset exists that has a lower inconsistency rate, then the algorithm will not find it. Thus, given an inconsistency rate larger than that of the full data set, LVF must be considered as a Monte Carlo algorithm, regardless of the number of iterations performed.

Given:

Examples $\mathbf{X} = < \mathbf{x}_1, y_1 >, < \mathbf{x}_M, y_M >, \ldots$
Number of iterations t_{max}
Number of prototypes p

Algorithm:

$\mathbf{X}_{best} \leftarrow$ random selection of p examples from $\mathbf{X}$
for $i \leftarrow 1$ **to** t_{max} **do**
  $\mathbf{X}' \leftarrow$ random selection of p examples from $\mathbf{X}$
  if $k\text{NN}(\mathbf{X}', \mathbf{X}) > k\text{NN}(\mathbf{X}_{best}, \mathbf{X})$ **then**
    $\mathbf{X}_{best} \leftarrow \mathbf{X}'$
return$(\mathbf{X}_{best})$

FIGURE 1.3: The Monte Carlo 1 algorithm [19].

1.6.2 Two Simple Monte Carlo Approaches

Consider next two applications of Monte Carlo randomization to feature selection. The goal of the first is to reduce the computational requirements of the nearest neighbor learner by sampling over the set of available training examples. The algorithm, called MC1 [19], repeatedly samples the data set in an attempt to find a small subset of prototypes (training examples) which allow nearest neighbor to generalize well to unseen examples.

The MC1 procedure begins by selecting p prototypes at random from the available examples, where p is chosen in advance by the user. Classification accuracy for nearest neighbor is then computed over the entire training set. If the selected set of examples leads to higher accuracy than the previous best subset, then the new subset is retained. This procedure is repeated t_{max} times, where t_{max} is also specified in advance by the user. The example subset which yields the highest accuracy is then returned at the end of the procedure and used on test data. Figure 1.3 summarizes the MC1 algorithm.

Notice that if we set t_{max} sufficiently large, then we are virtually guaranteed to find the best possible set of prototypes for a given value of p. Thus, like the LVF algorithm, MC1 behaves like a Las Vegas algorithm in the limit. Unlike LVF, which attempts to find the minimum number of features, MC1 does not necessarily find the minimum number of prototypes, p. Notice also that MC1 makes no assumptions particular to the nearest neighbor learner. The selection algorithm can therefore be adapted as a general purpose wrapper, and be used with any classification learning algorithm.

Skalak's experiments [19] show that MC1 performs best when the training and test data exhibit well defined class boundaries. Put another way, MC1 performs well when there is little overlap between examples from different classes. This may be an artifact of the nearest neighbor algorithm and not of Monte Carlo randomization in general. Nevertheless, the result reinforces the notion that we cannot expect randomized techniques to find a single specific

Given:

Examples $\mathbf{X} = < \mathbf{x}_1, y_1 >, \ldots < \mathbf{x}_m, y_m >$
Relevancy cut-off τ
Number of iterations t_{max}

Algorithm:

Partition $\mathbf{X}$ by class into $\mathbf{X}^+$ and $\mathbf{X}^-$
Initialize $\mathbf{w} = (0, 0, \ldots, 0)$
for $i \leftarrow 1$ **to** t_{max} **do** *//compute relevance*
  $\mathbf{x}_i \leftarrow$ random example $\mathbf{x} \in \mathbf{X}$
  $\mathbf{x}_i^+ \leftarrow$ nearest $\mathbf{x}^+ \in \mathbf{X}^+$ to $\mathbf{x}_i$
  $\mathbf{x}_i^- \leftarrow$ nearest $\mathbf{x}^- \in \mathbf{X}^-$ to $\mathbf{x}_i$
  if $\mathbf{x}_i \in \mathbf{X}^+$ **then**
    update$(\mathbf{w}, \mathbf{x}_i, \mathbf{x}_i^+, \mathbf{x}_i^-)$
  else
    update$(\mathbf{w}, \mathbf{x}_i, \mathbf{x}_i^-, \mathbf{x}_i^+)$
for $i \leftarrow 1$ **to** N **do** *//select most relevant*
  if $\frac{\mathbf{w}_i}{t_{max}} \geq \tau$ **then**
    feature i is relevant

Procedure update$(\mathbf{w}, \mathbf{x}, \mathbf{x}^+, \mathbf{x}^-)$ *//update relevance values*
for $i \leftarrow 1$ **to** N **do**
  $\mathbf{w}_i \leftarrow \mathbf{w}_i - \text{diff}(\mathbf{x}, \mathbf{x}^+)^2 + \text{diff}(\mathbf{x}, \mathbf{x}^-)^2$

FIGURE 1.4: The Relief algorithm [9].

solution within a large search space.

The Relief algorithm [9] demonstrates a different use of Monte Carlo randomization. Relief is a basic, two-class filtering algorithm which ranks variables according to a statistical measure of how well individual features separate the two classes. In an effort to reduce the computational cost of calculating these statistics, Relief selects examples at random for the computation.

Briefly, the algorithm operates by calculating a weight value for each of the N available features. These weights are calculated using a random sample of examples from the full set of supervised examples $\mathbf{X}$. Relief selects a training example $\mathbf{x}_i$ at random and then finds, according to Euclidean distance, the nearest same-class example $\mathbf{x}_i^+$ and the nearest different-class example $\mathbf{x}_i^-$. These examples are then used to update the weight value for each feature according to the difference between $\mathbf{x}_i$, $\mathbf{x}_i^+$, and $\mathbf{x}_i^-$. Here, the difference for nominal feature k is defined as 1 if $\mathbf{x}_{i,k}$ and $\mathbf{x}_{j,k}$ have different nominal values, and is defined as 0 if they are the same. For numerical features, the difference is simply $\mathbf{x}_{i,k} - \mathbf{x}_{j,k}$ normalized into the range $[0, 1]$. This procedure is repeated t_{max} times for some preset value of t_{max}. Features with weight greater than a specified value τ are considered relevant to the target output variable. Figure 1.4 summarizes the relief algorithm.

Notice the similarities and differences between MC1 and Relief. Both algorithms use randomization to avoid evaluating all M available training examples. MC1 achieves this goal by evaluating many hypotheses on different random example subsets, while Relief simply selects one random subset of examples on which to perform evaluations. Relief's approach is faster computationally, but cannot provide the user with any confidence that the selection of examples is representative of the sample space. In particular, the fewer examples selected, the less likely the random subset will provide a representative sample of the space. MC1 mitigates this problem by searching for the most beneficial, and presumably representative, example subset.

1.6.3 Random Mutation Hill Climbing

Skalak [19] discusses a feature selection approach based on randomized local search, called random mutation hill climbing (RMHC). As with the MC1 algorithm, the goal is to reduce the computational cost of the nearest neighbor learner while maximizing classification accuracy. Unlike MC1, which samples the space of possible prototype subsets, the RMHC algorithm conducts a more localized search by changing only one included prototype per iteration.

RMHC uses a single bit vector to encode the index of each of the p selected prototypes. This bit vector is initialized randomly, and the algorithm proceeds by flipping one randomly selected bit on each iteration. This has the effect of replacing exactly one prototype with another. The new set of prototypes is then evaluated on the entire training set using nearest neighbor, and is retained if it produces higher accuracy than the current set. Otherwise the change is discarded. The algorithm terminates after a fixed number of iterations, t_{max}. Figure 1.5 summarizes the RMHC algorithm. Note that, like MC1, RMHC can be adapted for use with learning algorithms other than nearest neighbor.

Skalak also describes a variant of the algorithm in which the bit vector also encodes which of the features are selected for use. Here, when a bit is selected for flipping, it may either change the set of included prototypes, or the set of included features. No control over the relative probability of these changes is considered. Experimental results, though limited, suggest that RMHC does improve both the computational requirements and the classification performance of k-nearest neighbor. Notice however, that because the random selections are embedded in a greedy local search, RMHC does not necessarily avoid falling into local extrema. Thus, RMHC is a Monte Carlo algorithm that cannot be converted into a Las Vegas algorithm simply by increasing the number of iterations, t_{max}. We can still convert RMHC to a Las Vegas algorithm by running the algorithm many times, however.

Given:

Examples $\mathbf{X} = < \mathbf{x}_1, y_1 >, \ldots, < \mathbf{x}_m, y_m >$
Number of iterations t_{max}
Number of prototypes p

Algorithm:

$\mathbf{X}_{best} \leftarrow$ random selection of p examples from $\mathbf{X}$
$\mathbf{b} \leftarrow$ random bit vector encoding p prototype indicies
for $i \leftarrow 1$ **to** t_{max} **do**
  $j \leftarrow$ random number between $0 \ldots |\mathbf{b}|$
  flip bit $\mathbf{b}_j$
  $\mathbf{X}' \leftarrow$ set of prototypes from $\mathbf{X}$ included by $\mathbf{b}$
  if $k\text{NN}(\mathbf{X}', \mathbf{X}) > k\text{NN}(\mathbf{X}_{best}, \mathbf{X})$ **then**
    $\mathbf{X}_{best} \leftarrow \mathbf{X}'$
return$(\mathbf{X}_{best})$

FIGURE 1.5: The random mutation hill climbing algorithm [19].

1.6.4 Simulated Annealing

Simulated annealing [10, 2] is a general purpose stochastic search algorithm inspired by a process used in metallurgy. The heating and slow cooling technique of annealing allows the initially excited and disorganized atoms of a metal to find strong, stable configurations. Likewise, simulated annealing seeks solutions to optimization problems by initially manipulating the solution at random (high temperature), and then slowly increasing the ratio of greedy improvements taken (cooling) until no further improvements are found.

To apply simulated annealing, we must specify three parameters. First is an annealing schedule, which consists of an initial and final temperature, T_0 and T_{final}, along with an annealing (cooling) constant ΔT. Together these govern how the search will proceed over time and when the search will stop. The second parameter is a function used to evaluate potential solutions (feature subsets). The goal of simulated annealing is to optimize this function. For this discussion, we assume that higher evaluation scores are better. In the context of feature selection, relevant evaluation functions include the accuracy of a given learning algorithm using the current feature subset (creating a wrapper algorithm), or a variety of statistical scores (producing a filter algorithm).

The final parameter for simulated annealing is a neighbor function, which takes the current solution and temperature as input, and returns a new, "nearby" solution. The role of the temperature is to govern the size of the neighborhood. At high temperature the neighborhood should be large, allowing the algorithm to explore broadly. At low temperature, the neighborhood should be small, forcing the algorithm to explore locally. For example, suppose we represent the set of available features as a bit vector, such that each bit indicates the presence or absence of a particular feature. At high temper-

Given:

Examples $\mathbf{X} =< \mathbf{x}_1, y_1 >, \ldots < \mathbf{x}_m, y_m >$
Annealing schedule, T_0, T_{final} and ΔT with $0 < \Delta T < 1$
Feature subset evaluation function $Eval(\cdot, \cdot)$
Feature subset neighbor function $Neighbor(\cdot, \cdot)$

Algorithm:

$S_{best} \leftarrow$ random feature subset
while $T_i > T_{final}$ **do**
  $S_i \leftarrow Neighbor(S_{best},\ T_i)$
  $\Delta E \leftarrow Eval(S_{best},\ \mathbf{X}) - Eval(S_i,\ \mathbf{X})$
  if $\Delta E < 0$ **then** *//if new subset better*
    $S_{best} \leftarrow S_i$
  else *//if new subset worse*
    $S_{best} \leftarrow S_i$ with probability $\exp(\frac{-\Delta E}{T_i})$
  $T_{i+1} \leftarrow \Delta T \times T_i$
return(S_{best})

FIGURE 1.6: A basic simulated annealing algorithm.

ature, the neighbor function may flip many bits to produce the next solution, while at low temperature the neighbor function may flip just one bit.

The simulated annealing algorithm, shown in Figure 1.6, attempts to iteratively improve a randomly generated initial solution. On each iteration, the algorithm generates a neighboring solution and computes the difference in quality (energy, by analogy to metallurgy process) between the current and candidate solutions. If the new solution is better, then it is retained. Otherwise, the new solution is retained with a probability that is dependent on the quality difference, ΔE, and the temperature. The temperature is then reduced for the next iteration.

Success in simulated annealing depends heavily on the choice of the annealing schedule. If ΔT is too large (near one), the temperature decreases slowly, resulting in slow convergence. If ΔT is too small (near zero), then the temperature decreases quickly and convergence will likely reach a local extrema. Moreover, the range of temperatures used for an application of simulated annealing must be scaled to control the probability of accepting a low-quality candidate solution. This probability, computed as $\exp(\frac{-\Delta E}{T_i})$, should be large at high temperature and small at low temperature to facilitate exploration early in the search, and greedy choices later in the search.

In spite of the strong dependence on the cooling schedule, simulated annealing is guaranteed to converge provided that the schedule is sufficiently long [6]. From a theoretical point of view, simulated annealing is therefore a Las Vegas algorithm. However, in practice the convergence guarantee requires intractably long cooling schedules, resulting in a Monte Carlo algorithm. Al-

though the literature contains relatively few examples of simulated annealing applications to feature selection, the extent to which the algorithm can be customized (annealing schedule, neighbor function, evaluation function) makes it a good candidate for future work. As noted above, simulated annealing supports both wrapper and filter approaches to feature selection.

1.6.5 Genetic Algorithms

Like simulated annealing, genetic algorithms are a general purpose mechanism for randomized search. There are four key aspects to their use: encoding, population, operators, and fitness. First, the individual states in the search space must be encoded into some string-based format, typically bit-strings similar to those used by RMHC. Second, an initial population of individuals (search states, such as feature subsets) must be selected at random. Third, one or more operators must be defined as a method for exchanging information among individuals in the population. Operators define how the search proceeds through state space. Typical operators include crossover, which pairs two individuals for the exchange of substrings, and mutation, which changes a randomly selected bit in an individual string with low probability. Finally, a fitness function must be defined to evaluate the quality of states in the population. The goal of genetic algorithms is to optimize the population with respect to the fitness function.

The search conducted by a genetic algorithm proceeds iteratively. Individuals in the population are first selected probabilistically with replacement based on their fitness scores. Selected individuals are then paired and crossover is performed, producing two new individuals. These are next mutated with low probability and finally injected into the next population. Figure 1.7 shows a basic genetic algorithm.

Genetic algorithms have been applied to the feature selection problem in several different ways. For example, Vafaie and De Jong [21] describe a straightforward use of genetic algorithms in which individuals are represented by bit-strings. Each bit marks the presence or absence of a specific feature. The fitness function then evaluates individuals by training and then testing a specified learning algorithm based on only the features that the individual specifies for inclusion.

In a similar vein, SET-Gen [3] uses a fitness function that includes both the accuracy of the induced model and the comprehensibility of the model. The learning model used in their experiments was a decision tree, and comprehensibility was defined as a combination of tree size and number of features used. The FSS-EBNA algorithm [7] takes a more complex approach to crossover by using a Bayesian network to mate individuals.

Two well-known issues with genetic algorithms relate to the computational cost of the search and local minima in the evaluation function. Genetic algorithms maintain a population (100 is a common size) of search space states which are mated to produce offspring with properties of both parents. The ef-

Given:

Examples $\mathbf{X} = < \mathbf{x}_1, y_1 >, \ldots < \mathbf{x}_m, y_m >$
Fitness function $f(\cdot, \cdot)$
Fitness threshold τ
Population size p

Algorithm:

$P_0 \leftarrow$ population of p random individuals
for $k \leftarrow 0$ **to** ∞ **do**
  $sum \leftarrow 0$
  for each individual $i \in P_k$ **do** *//compute pop fitness*
    $sum \leftarrow sum + f(i,\ \mathbf{X})$
    if $f(i,\ \mathbf{X}) \geq \tau$ **then**
      return(i)
  for each individual $i \in P_k$ **do** *//compute selection probs*
    $\Pr_k[i] \leftarrow \frac{f(i,\ \mathbf{X})}{sum}$
  for $j \leftarrow 1$ **to** $\frac{p}{2}$ **do** *//select and breed*
    select $i_1, i_2 \in P_k$ according to $\Pr_k$ with replacement
    $i_1, i_2 \leftarrow crossover(i_1, i_2)$
    $i_1 \leftarrow mutate(i_1)$
    $i_2 \leftarrow mutate(i_2)$
    $P_{k+1} \leftarrow P_{k+1} + \{i_1, i_2\}$

FIGURE 1.7: A basic genetic algorithm.

fect is an initially broad search that targets more specific areas of the space as search progresses. Thus, genetic algorithms tend to drift through the search space based on properties of individuals in the population. A wide variety of states, or feature subsets in this case, are explored. However, the cost of so much exploration can easily exceed the cost of a traditional greedy search.

The second problem with genetic algorithms occurs when the evaluation function is non-monotonic. The population may quickly focus on a local maximum in the search space, and become trapped. The mutation operator, a broad sampling of the state space in the initial population, and several other tricks are known to mitigate this effect. Goldberg [5] and Mitchell [15] provide detailed discussions of the subtleties and nuances involved in setting up a genetic search. Nevertheless, there is no guarantee that genetic algorithms will produce the best, or even a good result. This issue may arise with any probabilistic algorithm, but some are more prone to becoming trapped in suboptimal solutions than others.

1.6.6 Randomized Variable Elimination

Each of the algorithms considered so far use a simple form of randomization to explore the space of feature or example subsets. MC1 and LVF both sample the space of possible subsets globally, while RMHC samples the space in the context of a greedy local search. Simulated annealing and genetic algorithms, meanwhile, conduct initially broad searches that incrementally target more specific areas over time. The next algorithm we consider samples the search space in a more directed manner.

Randomized variable elimination (RVE) [20] is a wrapper method motivated by the idea that, in the presence of many irrelevant variables, the probability of selecting several irrelevant variables simultaneously at random is high. RVE searches backward through the space of variable subsets, attempting to eliminate one or more variables per step. Randomization allows for selection of irrelevant variables with high probability, while selecting multiple variables allows the algorithm to move through the space without incurring the cost of evaluating the intervening points in the search space. RVE conducts its search along a very narrow trajectory, sampling variable subsets sparsely, rather than broadly and uniformly. This more structured approach allows RVE to reduce substantially the total cost of identifying relevant variables.

A backward search serves two purposes for this algorithm. First, backward elimination eases the problem of recognizing irrelevant or redundant variables. As long as a core set of relevant variables remains intact, removing other variables should not harm the performance of a learning algorithm. Indeed, the learner's performance may increase as irrelevant features are removed from consideration. In contrast, variables whose relevance depends on the presence of other variables may have no noticeable effect when selected in a forward manner. Thus, mistakes should be recognized immediately via backward elimination, while good selections may go unrecognized by a forward selection algorithm.

The second purpose of backward elimination is to ease the process of selecting variables for removal. If most variables in a problem are irrelevant, then a random selection of variables is likely to uncover them. Conversely, a random selection is unlikely to turn up relevant variables in a forward search. Thus, forward search must work harder to find each relevant variable than backward search does for irrelevant variables.

RVE begins by executing the learning algorithm $\mathcal{L}$ on data which includes all N variables. This generates an initial hypothesis h. Next, the algorithm selects k input variables at random for removal. To determine the value of k, RVE computes a cost (with respect to a given learning algorithm) of attempting to remove k input variables out of n remaining variables given that r are relevant. Note that knowledge of r is required here, although the assumption is later removed. A table $k_{opt}(n, r)$ of values for k given n and r is then computed via dynamic programming by minimizing the aggregate cost of removing all $N - r$ irrelevant variables. Note that n represents the

Given:

Examples $\mathbf{X} = < \mathbf{x}_1, y_1 >, \ldots < \mathbf{x}_m, y_m >$
Learning algorithm $\mathcal{L}$
Number of input features N
Number of relevant features r

Algorithm:

$n \leftarrow N$
$h \leftarrow$ hypothesis produced by $\mathcal{L}$ on all N inputs
compute schedule $k_{opt}(i, r)$ for $r < i \leq N$ by dynamic programming
while $n > r$ **do**
    select $k_{opt}(n, r)$ variables at random and remove them
    $h' \leftarrow$ hypothesis produced by $\mathcal{L}$ on $n - k_{opt}(n, r)$ inputs
    if $error(h', \mathbf{X}) < error(h, \mathbf{X})$ **then**
        $n \leftarrow n - k_{opt}(n, r)$
        $h \leftarrow h'$
    else
        replace the $k_{opt}(n, r)$ selected variables
return(h)

FIGURE 1.8: The randomized variable elimination algorithm [20].

number of remaining variables, while N denotes the total number of variables in the original problem.

On each iteration, RVE selects $k_{opt}(n, r)$ variables at random for removal. The learning algorithm is then trained on the remaining $n - k_{opt}(n, r)$ inputs, and a hypothesis h' is produced. If the error $e(h')$ is less then the error of the previous best hypothesis $e(h)$, then the selected inputs are marked as irrelevant and are all simultaneously removed from future consideration. If the learner was unsuccessful, meaning the new hypothesis had larger error, then at least one of the selected inputs must have been relevant. The removed variables are replaced, a new set of $k_{opt}(n, r)$ inputs is selected, and the process repeats. The algorithm terminates when all $N - r$ irrelevant inputs have been removed. Figure 1.8 shows the RVE algorithm.

Analysis of RVE [20] shows that the algorithm expects to evaluate only $O(r \log(N))$ variable subsets to remove all irrelevant variables. This is a striking result, as it implies that a randomized backward selection wrapper algorithm evaluates fewer subsets, and requires less total computation than forward selection wrapper algorithms. Stracuzzi and Utgoff provide a detailed formal analysis of randomized variable elimination [20].

The assumption that the number of relevant variables r is known in advance plays a critical role in the RVE algorithm. In practice, this is a strong assumption that is not typically met. Stracuzzi and Utgoff [20] therefore provide a version of the algorithm, called RVE*r*S (pronounced "reverse"), that

conducts a binary search for r during RVE's search for relevant variables.

Experimental studies suggest RVErS evaluates a sublinear number of variable subsets for problems with sufficiently many variables. This conforms to the performance predicted by the analysis of RVE. Experiments also show that for problems with hundreds or thousands of variables, RVErS typically requires less computation than a greedy forward selection algorithm while producing competitive accuracy results. In practice, randomized variable elimination is likely to be effective for any problem which contains many irrelevant variables.

1.7 Issues in Randomization

The preceding sections in this chapter covered the basic use of randomization as an algorithmic technique, specifically as applied to feature selection. We now consider more advanced issues in applying randomization. Of particular interest and importance are sampling techniques, and the source of the random numbers used in the algorithms.

1.7.1 Pseudorandom Number Generators

Randomized algorithms necessarily depend on the ability to produce a sequence of random numbers. However, deterministic machines such as modern computers are not capable of producing sequences of truly random numbers. John von Neumann once stated that, "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin" [17]. In practice, we must rely on pseudorandom number generators to provide sequences of numbers that exhibit statistical properties similar to those of genuinely random numbers.

The main property of pseudorandom numbers that differs from true random numbers is periodicity. No matter how sophisticated a pseudorandom number generating algorithm may be, it will eventually revisit a past state and begin repeating the number sequence. Other possible problems with pseudorandom number generators include non-uniform distribution of the output sequence, correlation of successive values (predictability), and short periods for certain starting points. The presence of any of these properties can cause poor or unexpected performance in a randomized algorithm.

The primary defense against such undesirable results is to select a good pseudorandom number generator prior to running any experiments. For example, the Mersenne twister algorithm [14] has proved useful for statistical simulations and generative modeling purposes. The algorithm has a very long period of 2^{19937}, provides a provably good distribution of values, and is

computationally inexpensive. A variety of other suitable, but less complex algorithms are also available, particularly if the user knows in advance that the length of the required sequence of pseudorandom numbers is short.

1.7.2 Sampling from Specialized Data Structures

A second possible pitfall in the use of randomized algorithms stems from sampling techniques. For small databases, such as those that can be stored in a simple table or matrix, examples and features (rows and columns respectively) may be selected by simply by picking an index at random. However, many large databases are stored in more sophisticated, non-linear data structures. Uniformly distributed, random samples of examples cannot be extracted from such databases via simple, linear sampling methods.

An improperly extracted sample is unlikely to be representative of the larger database. The results of a feature selection or other learning algorithm run on such a sample may not extrapolate well to the rest of the database. In other words, the error achieved by feature selection and/or learning algorithm on a sampled test database will be overly optimistic. In general, different data structures will require different specialized sampling methods.

One example of a specialized sampling algorithm is discussed by Makawita, Tan and Liu [13]. Here, the problem is to sample uniformly from a search tree that has a variable number of children at internal nodes. The naive approach of simply starting at the root and then selecting random children at each step until reaching a leaf (known as a random walk) will tend to oversample from paths that have few children at each internal node. This is an artifact of the data structure and not the data itself, and so is unacceptable. The presented solution is to bias the acceptance of the leaf node into the sample by keeping track of the fanout at each internal node along the path. Leaves from paths with low fanout are accepted with lower probability than those from paths with high fanout. The sampling bias of the naive algorithm is thus removed.

1.8 Summary

The feature selection problem possesses characteristics that are critical to successful applications of randomization. First, the space of all possible feature subsets is often prohibitively large. This means that there are many possible choices available at each step in the search, such as which feature to include or exclude next. Second, those choices are often difficult to evaluate, because learning algorithms are expensive to execute and there may be complex interdependencies among features. Third, deterministic selection algorithms are often prone to becoming trapped in local optimal, also due

to interdependencies among features. Finally, there are often too many examples available for an algorithm to consider each one deterministically. A randomized approach of sampling feature subset space helps to mitigate all of these circumstances.

In practice, there are several important issues to consider when constructing a randomized algorithm for feature selection. First is the decision of which aspect of the problem will be randomized. One option is to randomize over the set of input variables, causing the resulting algorithm to search for the best variable subset by sampling from the space of all subsets. A second approach is to randomize over the set of training examples, creating an algorithm that considers only a portion of the available training data. Finally, one may also randomize over both the input variables and the training data. In any case, the achieved reduction in computational cost must be balanced against a loss of confidence in the solution.

The second issue to consider in randomization relates to the performance of the resulting algorithm. Some tasks may demand discovery of the best possible feature subset, necessitating use of a Las Vegas algorithm. Other tasks may sacrifice solution quality for speed, making a Monte Carlo algorithm more appropriate. A third option is to generate an initial solution, and then improve the solution over time, as in anytime algorithms [22]. Many more specific guarantees on performance are also possible.

Other issues in the application of randomization include the quality of the pseudorandom number generator used, and the sampling technique that is used. Both of these can impact the performance of the randomized algorithm. The feature selection literature contains examples of the different randomizations methods (randomization over features versus examples), a variety of performance guarantees, and special purpose sampling methods, as discussed throughout the chapter. Although far from a complete exposition, this chapter should provide sufficient information to launch further study of randomized algorithms in the context of feature selection.

References

[1] R. Caruana and D. Freitag. Greedy attribute selection. In *Machine Learning: Proceedings of the Eleventh International Conference*, pages 121–129, New Brunswick, NJ, 1994. Morgan Kaufmann.

[2] V. Černý. A thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[3] K. Cherkauer and J. Shavlik. Growing simpler decision trees to facilitate knowledge discovery. In E. Simoudis, J. Han, and U. M. Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, Portland, OR, 1996. AAAI Press.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[5] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[6] H. M. Hastings. Convergence of simulated annealing. *ACM SIGACT News*, 17(2):52–63, 1985.

[7] I. Inza, P. Larranaga, E. R., and B. Sierra. Feature subset selection by Bayesian network-based optimization. *Artificial Intelligence*, 123(1–2):157–184, 2000.

[8] G. H. John, R. Kohavi, and K. Pfleger. Irrelevant features and the subset selection problem. In *Machine Learning: Proceedings of the Eleventh International Conference*, pages 121–129, New Brunswick, NJ, 1994. Morgan Kaufmann.

[9] K. Kira and L. Rendell. A practical approach to feature selection. In S. D. H. and P. Edwards, editors, *Machine Learning: Proceedings of the Ninth International Conference*, Aberdeen, Scotland, UK, 1992. Morgan Kaufmann.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[11] W. Krauth. Introduction to monte carlo algorithms. In J. Kertesz and I. Kondor, editors, *Advances in Computer Simulation*, Lecture Notes in Physics. SpringerVerlag, New York, 1998.

0-8493-0052-5/00/$0.00+$.50

[12] H. Liu and R. Setino. A probabilistic approach to feature selection. In L. Saitta, editor, *Machine Learning: Proceedings of the Thirteenth International Conference on Machine Learning*, pages 319–327, Bari, Italy, 1996. Morgan Kaufmann.

[13] D. P. Makawita, K.-L. Tan, and H. Liu. Sampling from databases using B^+-trees. In *Proceedings of the 2000 ACM CIKM International Conference on Information and Knowledge Management*, pages 158–164, McLean, VA, 2000. ACM.

[14] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3 – 30, 1998.

[15] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.

[16] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, UK, 1995.

[17] J. von Neumann. Various techniques used in connection with random digits. In *Applied Mathematics Series, no. 12*. 1951.

[18] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.

[19] D. B. Skalak. Prototype and feature selection by sampling and random mutation hill climbing. In W. W. Cohen and H. Hirsh, editors, *Machine Learning: Proceedings of the Eleventh International Conference*, pages 293–301, New Brunswick, NJ, 1994. Morgan Kaufmann.

[20] D. J. Stracuzzi and P. E. Utgoff. Randomized variable elimination. *Journal of Machine Learning Research*, 5:1331–1364, 2004.

[21] H. Vafaie and K. DeJong. Genetic algorithms as a tool for restructuring feature space representations. In *Proceedings of the Seventh International Conference on Tools with AI*, Herndon, VA, 1995. IEEE Computer Society Press.

[22] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3), 1996.

Index