3 FEATURE SELECTION ASPECTS

With a unified model of feature selection, we are ready to discuss in detail different aspects of feature selection. The major aspects of feature selection are (1) search directions (feature subset generation), (2) search strategies, and (3) evaluation measures. The objective of this chapter is two-fold: (a) to study the various options for each aspect in a systematic and principled way and (b) to identify the essential and different characteristics of various feature selection systems.

3.1 OVERVIEW

The study of different perspectives of feature selection manifests that search and measure play dominant roles in feature selection; stopping criteria are usually determined by a particular combination of search and measure. Since search is about search directions and search strategies, with evaluation measures, we come up with a 3-dimensional structure, in Figure 3.1, that categorizes all possible feature selection algorithms in terms of search and measure.

There are 27 possible combinations of feature selection methods, considering all the possibilities. We will elaborate on each possibility along every dimension

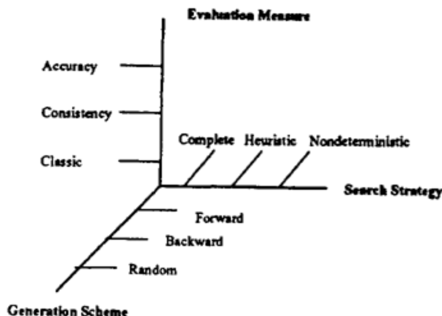

Figure 3.1. Three principal dimensions of feature selection: search strategy, evaluation measure, and feature generation scheme.

later in this chapter and on individual combinations in Chapter 4. From the point of view of a method's output, however, these methods can be grouped into just two categories. One category is about ranking features according to some evaluation criterion; the other is about choosing a minimum set of features that satisfies an evaluation criterion. We first discuss these two categories.

Feature Ranking Algorithms. In this category of feature selection algorithms, one can expect a ranked list of features which are ordered according to evaluation measures. A measure can be any of accuracy, consistency, information, distance, and dependence. In other words, an algorithm of this type does not tell you what is the minimum set of features, but only the importance (relevance) of a feature compared to others.

The idea is to evaluate each individual feature with a measure. This evaluation results in a value attached to a feature. Features are then sorted according to the values. The run time complexity of this simple algorithm (summarized in Table 3.1) is $O(N \times n + N^2)$ where N is the number of features, n the number of instances: $O(n)$ for step (1); $O(N)$ for step (2); and the **for** loop repeats N times. Many variations of this algorithm lead to different feature selection methods. What is common to these feature ranking methods is a ranked list of features.

One can simply choose the first M features for the task at hand if he knows what M is. It may not be so straightforward if M is unknown. Variants of this algorithm have been designed to overcome the unknown M problem. One

Table 3.1. A univariate feature ranking algorithm.

Ranking Algorithm
Input: $\mathbf{x}$ - features, U - measure

initialize: list $L = \{\}$ /* *L stores ordered features* */
for each feature x_i, $i \in \{1, \ldots, N\}$
begin
(1) $v_i =$ `compute`(x_i,U)
(2) position x_i into L according to v_i
end

Output: L in which the most relevant feature is placed first.

solution is to use the evaluation measure repeatedly to build a decision tree classifier (Section 2.4.2 of Chapter 2); the features used in the decision tree are selected. This variation can solve the unknown M problem. This treatment conceptually blurs the demarcation between feature ranking algorithms and minimum subset algorithms. Some researchers did use this method to select features. However, it may not be wise to use this on the data in which some irrelevant feature is strongly correlated to the class feature (John et al., 1994). It is not recommended by (Almuallim and Dietterich, 1994) if the goal is to find the minimum feature subset.

Minimum Subset Algorithms. It is quite often that one does not know the number of relevant features. In order to search for it, one may need to design other types of algorithms since feature ranking algorithms can only order features. Thus we have this category of minimum subset algorithms. An algorithm in this category returns a minimum feature subset and no difference is made for features in the subset. Whatever in the subset are relevant, otherwise irrelevant.

The **Min-Set** algorithm can be found in Table 3.2. The **`subsetGenerate()`** function returns a subset following a certain search method, in which a stopping criterion determines when **`stop`** is set to *true*; function **`legitimacy()`** returns *true* if subset S_k satisfies measure U. Function **`subsetGenerate()`** can take one of the generation schemes.

Here we introduce two types of algorithms from the viewpoint of the end results of feature selection. The details vary when we discuss each of the three dimensions (generation scheme, search strategy, and evaluation measure) for feature selection. In the following sections, we will elucidate each possibility

Table 3.2. A minimum subset algorithm. # - a cardinality operator returns the number of features in a set.

```
Min-Set Algorithm
  Input: x - features, U - measure

  initialize: S = {}                    /* S holds the minimum set*/
              stop = false;
  repeat
    (1) S_k = subsetGenerate(x)         /* stop can be set here*/
        if
    (2)     legitimacy(S_k, U) is true and #(S_k) < #(S)
        then
    (3)     S = S_k                     /* S is replaced by S_k*/
  until stop = true

  Output: S - the minimum legitimate subset of features.
```

along a single dimension (focusing on one dimension at a time) by giving algorithms and examples.

3.2 BASIC FEATURE GENERATION SCHEMES

We now look at the ways in which feature subsets are generated. For instance, one can start with an empty set, and fill in the set by choosing the most relevant features from the original features. Or one can begin with the original full set, and remove irrelevant features. Among many variations, we choose three basic schemes (sequential forward, sequential backward and random) plus a combination of the first two, and others can be obtained by varying some parts of these basic schemes.

3.2.1 Sequential forward generation

This scheme starts with an empty set S and adds features from the original set F into S sequentially, i.e., one by one. Obviously, if one wants to obtain a ranked list, this scheme can provide that. In Table 3.3, some algorithmic details are given. The idea is *sequential forward generation* (hence **SFG**). At each round of selection, the best feature f is chosen by `FindNxt`(F). f is added into S and removed from F. So, S grows and F shrinks. There are two stopping criteria. If one wants to have a minimum feature subset, he can rely

on the first stopping criterion "S satisfies U". The generation terminates at the moment when S satisfies U - the evaluation measure. S is the minimum subset according to U. If one would like to have a ranked list, he can use only the second stopping criterion "$F = \{\}$". As we know, the first feature chosen is the most relevant, and the last feature chosen the least relevant. By adopting only the second stopping criterion, one can indeed obtain a ranked list. The second stopping criterion ensures that the algorithm should stop when the evaluation measure U cannot be satisfied even if all attributes are chosen by S (therefore F is empty). This is quite often when noise is present in the data.

Table 3.3. Sequential forward feature set generation - **SFG**.

SFG Scheme
Input: F - full set, U - measure

initialize: $S = \{\}$ /* *S stores the selected features* */
repeat
(1) $f =$ `FindNxt`(F)
(2) $S = S \cup \{f\}$
(3) $F = F - \{f\}$
until S satisfies U or $F = \{\}$

Output: S

SFG is just a simple scheme for forward feature set generation. Other forms can be easily obtained by varying statement (1) $f =$ `FindNxt`(F). For instance, if one suspects that two features together may play a more influential role (individually they may not), then he may modify `FindNxt()` into `FindNxt2()`, or finding up to the best two features at a time (Liu and Wen, 1993), i.e., having one more step of look-ahead. In order to apply `FindNxt2()`, we need to try $\binom{N}{1}$ plus $\binom{N}{2}$ combinations in order to find the best one or two features. Let's save the chosen features in S_1, the remaining features in S_2. Then, we choose from S_2 another one or two features in order to find next best features, which is determined by the value of U after combining these newly selected features with the ones in S_1. In theory, one can go up to N-step look-ahead and check all the combinations in order to choose the best subset. The more the steps of look-ahead, generally, the better a solution is. Nevertheless, that would take enormous computing time since the running time needed is exponentially proportional to the number of features N, or $O(2^N)$. Hence, the one-step look-ahead version of **SFG** is among the most commonly used schemes in subset generation because of its efficiency.

3.2.2 Sequential backward generation

This is a scheme which is quite an opposite of **SFG**. As shown in Table 3.4, the search starts with the full set F and finds a feature subset S by removing one feature at a time, so is named as **SBG** - sequential backward generation. Instead of finding the most relevant feature, `GetNxt`(F) looks for the least relevant feature f which is then removed from F. Although, it is the same as in **SFG** that F shrinks and S grows, but it is F that is the subset sought out. S only stores the irrelevant features, which may or may not be useful. If the full set is the minimum set itself, **SBG** will identify this in one loop. Otherwise, $F \cup \{f\}$ output by **SBG** is the minimum set that can satisfy U since F cannot satisfy U. The second stopping criterion "$F = \{\}$" ensures that the algorithm should stop if all instances in the data have the same class value. One may notice that unless F is empty, features in F are not ranked. In other words, we have no idea which features are more relevant than the other. Hence, it is not a suitable scheme that gives a ranked list, although features in S are ranked according to their irrelevancy.

Table 3.4. Sequential backward feature generation - **SBG**.

SBG Scheme
Input: F - full set, U - measure

initialize: $S = \{\}$ /* *S holds the removed features* */
repeat
(1) $f =$ `GetNxt`(F)
(2) $F = F - \{f\}$
(3) $S = S \cup \{f\}$
until F does not satisfies U or $F = \{\}$

Output: $F \cup \{f\}$

In the same spirit of `FindNxt2`(), one may try `GetNxt2`() or its other variations. The similar run time complexity analysis applies, and **SFG** and **SBF's** run time complexities are of the same order. Because of the nature of sequential removal, the minimal subset obtained in this scheme may not be the absolute minimal subset (i.e., the optimal one). One way to get the absolute minimal subset is by trying all the combinations.

3.2.3 *Bidirectional generation*

As we discussed briefly in Chapter 2, **SFG** and **SBG** complement each other. When the number of relevant features is smaller than $N/2$, **SFG** is quicker, if the number of relevant features (M) is greater than $N/2$, then **SBG** is faster. Without knowing the value of M, one would not have a clue which scheme between the two is better. Hence, born is this bidirectional generation scheme, shown in Table 3.5. The basic implementation of this scheme (**FBG** - forward and backward generation) is running schemes **SFG** and **SBG** in parallel. **FBG** stops if either (**SFG** or **SBG**) finds a satisfactory subset. A flag is needed to tell which one has found an optimal subset.

Table 3.5. Bidirectional feature set generation - **FBG**.

FBG Scheme

Input: F_f, F_b - full set, U - measure

initialize: $S_f = \{\}$, /* S_f *holds the selected features**/
$S_b = \{\}$ /* S_b *holds the removed features.**/

repeat

(1) $f_f =$ `FindNxt`(F_f) $f_b =$ `GetNxt`(F_b)
(2) $S_f = S_f \cup \{f_f\}$ $F_b = F_b - \{f_b\}$
(3) $F_f = F_f - \{f_f\}$ $S_b = S_b \cup \{f_b\}$

until (a) S_f satisfies U or $F_f = \{\}$ or
(b) F_b does not satisfies U or $F_b = \{\}$

Output: S_f if (a) or $F_b \cup \{f_b\}$ if (b)

3.2.4 *Random generation*

FBG's putting two sequential generation schemes together can help in getting a valid subset fast on average, but cannot help in finding an *absolute* minimum valid set. The sheer reason is that all the above three schemes adopt a hill-climbing heuristic hoping that by selecting the best (as in **SFG**) or removing the worst (as in **SBG**) sequentially, an absolute minimal subset (optimal) will emerge. Doing so will surely speed up the selection process, but if they hit a local minimum (a best subset at that moment), they have no way to get out because what has been removed cannot be added (as in **SBG**) and what has been added cannot be removed (as in **SFG**). This is one fundamental problem with a sequential method. Put it another way, the three schemes above cannot

guarantee to find the optimal feature subset. As opposed to this fixed rule of sequential generation, we can resort to random feature generation, hoping to avoid getting stuck at some local minima. This random generation scheme produces subsets at random. Basically one needs a good random number generator and tailors it to RandGen(F) so that every combination of features F, ideally, has a chance to occur and occurs just once. Several implementations of random number generators can be found in (Press et al., 1992). This scheme is summarized as $S =$ RandGen(F) in Table 3.11 where S is a subset of features.

Although we introduce four schemes for feature generation in this section, there are three basic ones, the bidirectional feature generation is a combination of sequential forward and backward feature generation. Later on throughout the book, we consider the basic three schemes unless otherwise specified.

3.3 SEARCH STRATEGIES

In the preceding section, we mentioned an issue of optimal subsets when we discussed the feature generation schemes. This can be further elaborated in terms of search strategies. Roughly speaking, the search strategies can be of three types: (1) complete, (2) heuristic, and (3) nondeterministic. In this section, we will describe the characteristics of these search strategies and their usages. Then, for each search strategy, we take into account of the three basic feature generation schemes. As a consequence, we find a big variety of search methods at our disposal. In the following, we put feature selection into the perspective of search. The search space consists of all the combinations of feature subsets. The size of the search space is 2^N where N is the number of features.

3.3.1 Complete search

Exhaustive search is complete since it covers every combination of N features, $\binom{N}{1}, \binom{N}{2}, \ldots$ Under certain circumstances, search can be complete but not exhaustive. For instance, if an evaluation measure is monotonic, Branch & Bound (Narendra and Fukunaga, 1977) is complete search that guarantees an optimal subset. The difference between complete and exhaustive search is in that exhaustive search is complete, but complete search is not necessarily exhaustive. So we use complete search for both kinds of search. A measure U is monotonic if for any two subsets S_1, S_2, and $S_1 \subseteq S_2$, then $U(S_1) \geq U(S_2)$.

Two classic exhaustive search implementations are *depth-first search* and *breadth-first search*. Both types of search can be forward or backward in feature generation. In a forward generation scheme, it starts with an empty set, then considers the possible subsets of one feature, two features, etc. subsequently. This process is reversed for a backward generation scheme, starting with the

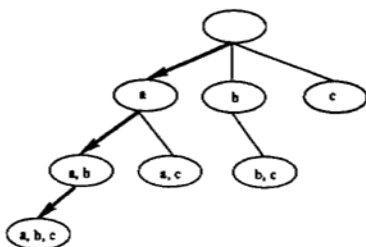

Figure 3.2a. Depth-first search illustration with three features a, b, c.

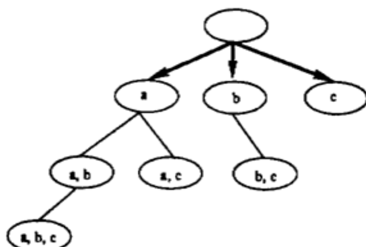

Figure 3.2b. Breadth-first search illustration with three features a, b, c.

full set of features. Regardless of their directions, the essence of the two types of search is the systematic examination of every possible subset. The difference is how the systematic search is carried out. We adopt the forward generation scheme in the explanation below. Assume that there are three features a, b, c in the full set F. We can avoid the duplicates of a subset by following an enumeration procedure. With the enumeration procedure, we can create a search tree instead of a search lattice. Since exhaustive search is under examination, we will visit all combinations of features (8 states including the empty set). As shown in Figures 3.2a and 3.2b, the depth-first search goes down one branch, backtracks to another branch until all braces are generated and checked, while the breadth-first search does it layer by layer, checking all subsets with one feature, then with two features, so on and so forth. Both search algorithms (**DEP** and **BRD**) are shown in Tables 3.5a, 3.5b and 3.6.

Two implementations of depth-first search are presented. One uses an explicit *stack* data structure; and the other employs implicit run-time stack. A stack is a last-in-first-out data structure. Breadth-first search is implemented using a *queue* data structure that enqueues subsets at each round of feature generation, dequeues the first subset in the queue for consideration to expand the search space. As in the example, the queue contains (a, b, c, ab, ac, bc, abc) if no node[1] is dequeued. When one works on a particular implementation, one should also give some consideration about the enumeration procedure. The working of the enumeration procedure is determined by the feature generation scheme, for instance, one needs to add one feature (choosing from F) at a time in the forward generation scheme, and to remove one feature at a time in the backward generation scheme. In Tables 3.5a and 3.5b, however, there is no explicit mentioning about the enumeration procedure for the sake of simplicity.

Table 3.5a. Exhaustive search: depth first - **DEP** with explicit use of a stack.

```
DEP Algorithm 1
  Input: F - full set, S - stack, U - measure
  initialize: node = null
              S = null
   DEP (node)
    {
     if node is the best subset so far w.r.t. U
       Set = node
     for all children C of node
        push (C, S)
     while (notEmpty(S)) {
       node = pop (S)
       DEP (node) }
    }
  Output: Set
```

Now we have a good picture of exhaustive search, let's look at an example of complete search: Branch & Bound search (Narendra and Fukunaga, 1977), and examine the difference between the two types of search. Branch & Bound is a variation of depth-first search of a lattice. Without any restriction, therefore, it is exhaustive search. With a given bound β, however, the search stops at a node whose measure is greater than β, and the branches extended from the node are pruned. The search backtracks and another branch will be searched. As seen in Figure 3.3, Branch & Bound usually adopts a backward feature generation

Table 3.5b. Exhaustive search: depth first - DEP without explicit use of a stack.

```
DEP Algorithm 2
  Input: F - full set, U - measure
  initialize: node = null
   DEP (node)
    {
     if node is the best subset so far w.r.t. U
       Set = node
     for all children C of node
       DEP (C)
    }
  Output: Set
```

Table 3.6. Exhaustive search: breadth first - BRD.

```
BRD Algorithm
  Input: F - full set, Q - queue, U - measure
  initialize: node = null
              Q = null
   BRD (node)
    {
     for all children C of node
        enqueue(C, Q)
     while (notEmpty(Q)) {
        node = dequeue(Q)
        if node is the best subset so far w.r.t. U
          Set = node
        BRD (node) }
    }
  Output: Set
```

scheme and begins with a full set of features. The success of Branch & Bound depends on a monotonic evaluation measure U. Let's take U as the number of errors made by a classifier. The evaluation of each subset S results in a value of U. If β is 12, as in the example shown in Figure 3.3, the search starts from the left-most branch, the value of subset (a, b) is 13, which is greater than β. Hence it backtracks to subset (a, c). Its value is 12, so the search continues.

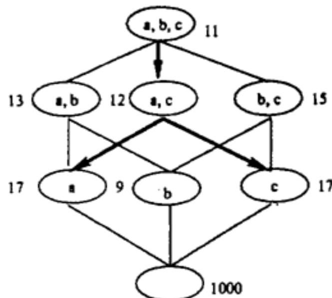

Figure 3.3. Branch & Bound search illustration with three features a, b, c. Numbers beside nodes are their costs.

Since the next two subsets (a) and (c) have higher values (17) the search stops, and (a, c) is recorded as the best subset so far found. The search continues to (b, c) but its value is greater than β, and the algorithm returns (a, c) as the best subset. As we see in the figure, the best subset should be (b) since it has the lowest U. Branch & Bound could not find it because the measure U used is obviously not monotonic (one case countering the monotonicity of U is that its value of subset (b, c) is greater than that of subset (c)). If U were monotonic, it would be guaranteed that the search is complete and subset (b) would be found. This example (Figure 3.3) illustrates the concept of Branch & Bound and shows the importance of a monotonic measure. The algorithm is described in Table 3.7. Later on, in Section "Evaluation Measures", various measures including monotonic ones will be introduced.

For exhaustive search, there is no parameter to be specified by a user. However, this situation changes when one tries to avoid exhaustive search and opts for complete search, more often than not, one needs to provide values for some parameters. In this Branch and Bound algorithm, for instance, the bound is supplied by the user. "How reasonable a bound is" plays a pivotal role. As is well known, the setting of values for parameters requires some sort of knowledge or educated guesses. In (Siedlecki and Sklansky, 1988), they showed some heuristic methods of finding a good bound. Later on, we will introduce another version of branch and bound which automatically sets the bound; the algorithm is named as ABB (Table 4.3). The reader may notice that the random feature generation scheme is not mentioned so far. This is because in discussing com-

Table 3.7. Complete search: Branch & Bound - BAB.

```
BAB Algorithm
  Input: F - full set, Q - queue, U - measure
         β - bound for some value of U

  initialize: node = F
              best = β
   BAB (node)
    {
      for all children C of node {
        if (C's value of U < β) {
                   /* else, the branch starting with C is pruned*/
           if (C's value ≤ best's value)
              best = C          /* remember the best subset*/
           BAB (C)
        }
      }
    }
   Output:  best                        /* the best set w.r.t. U */
```

plete search, it is natural for us to exclude any randomness as the search must be at least complete.

3.3.2 *Heuristic search*

Complete search strategies are usually time consuming. Can we make some smart choices based on the minimum information available, but without looking at the whole picture? This is all what heuristic search is about. The rationale of this non-optimal strategy is three-fold: (1) it is quick to find a solution (i.e., a subset of features); (2) it usually can find near optimal solution if not optimal; and (3) the trade-off of optimality with speed is often worthwhile because of much gained speed and little loss of optimality (recall that an optimal subset is a minimum one that satisfies the evaluation criterion). Heuristic feature selection algorithms abound. We introduce three of them (best-first, beam, and approximate branch & bound search) in relation with depth-first, breadth-first, and Branch & Bound.

Best-first search is derived from breadth-first search. Instead of branching on all nodes at each layer, best-first search expands its search space layer by layer, evaluates all *newly* generated subsets (the child nodes of the root in the

beginning), chooses *one best* subset at each layer to expand, and repeat this process until no further expansion is possible. Some researchers call this type of search myopic since it only cares what is the best at each step. In other words, it is a one-step look-ahead strategy. A natural modification is to look ahead a few more steps. Doing so may improve the quality of chosen subsets, but the run time will be increased exponentially with the increased steps of look-ahead. The reader can refer to the feature generation schemes such as `FindNxt()` and `FindNxt2()` in Section 3.2. The optimality of a subset is guaranteed only when exhaustive search is in place. Table 3.8 shows the algorithm for best-first search. An example is also shown in Figure 3.4 in which a path is depicted. It is clear that the run time complexity of best-first search is much lower than $O(2^N)$ where N is the total number of available features. As a matter of fact, the run time complexity is $O(mN)$ where m is the maximum number of children a node can have, and m is always smaller than N. A node A is a child node of another node B if and only if (1) $|\#(A) - \#(B)| = 1$ and (2) A is different from B in value by one feature. As in Figure 3.4, (a) is a child node of the empty set (the root), (a, b) is a child of (a), (a, b, c) is a child of (a, b), but not of (a) which is a grandparent node.

Table 3.8. Heuristic search: best-first - BEST.

BEST Algorithm
 Input: F - full set, Q - queue, U - measure
 initialize: $node$ = null
  Q = null
 BEST ($node$)
  {
  for all children C of $node$
   C_{best} is the best among all C's w.r.t. U
  enqueue(C_{best}, Q)
  while (notEmpty(Q)) {
   $node$ = dequeue(Q)
   if $node$ is the best subset so far w.r.t. U
    $Set = node$
   BEST ($node$) }
  }
 Output: Set

Beam search can be understood as an extension of best-first search, or a limited version of breadth-first search. In best-first search, we only choose the best subset at that moment and proceed further. How about choosing the best

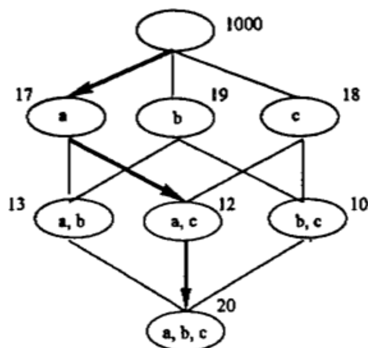

Figure 3.4. Best-first search illustration with three features a, b, c.

two subsets instead of one? In Figure 3.5, we show a case of beam search that chooses two best subsets to expand at every layer. As in Table 3.9, one can set the value of η to 2, 3, or up to N. Beam search is in effect breadth-first search when $\eta = N$. The reason we should consider more subsets is simple: we may not derive the best subset from the best two subsets at earlier steps (Cover, 1974). By relating beam search to breadth-first search, we clearly witness another example of trading time with optimality. Beam search's run time complexity is between best-first search and breadth-first search.

Approximate Branch & Bound is proposed by (Siedlecki and Sklansky, 1988) as an extension to Branch & Bound in the realization that it is rare to have a monotonic evaluation measure. The algorithm is regenerated in Table 3.10. As was shown in Figure 3.3, the best subset was missed due to the non-monotonicity of the measure. Can we still reach the best set? The answer is to relax bound β by δ so it allows Branch & Bound to continue the search when β is reached. In our example (first shown in Figure 3.3), now in Figure 3.6), if $\delta = 1$ (recall that $\beta = 12$), search does not stop at subset (a, b). This leads to the finding of the best subset (b) whose U is 9. At the first glance, one may think why not just take a larger β. It would achieve the same effect. However, conceptually, they are different. Introducing δ is a controlled relaxation. As we know that β is determined according to some prior knowledge, so after it is set, it should be left alone. δ is something linked with the monotonicity of a measure. When one suspects a measure's monotonicity, by varying δ, he can get a feeling about the measure. This is a heuristic way of overcoming

Table 3.9. Heuristic search: beam search - **BEAM**.

```
BEAM Algorithm
  Input: F - full set, Q - queue, U - measure
         η - a number of best children to evaluate
  initialize: node = null
         Q = null
  BEAM (node)
  {
    for all children C of node
       find η best C_best's among all C's w.r.t. U
    for all C_best's
       enqueue(C_best, Q)
    while (notEmpty(Q)) {
       node = dequeue(Q)
       if node is the best subset so far w.r.t. U
         Set = node
       BEAM (node) }
  }
  Output: Set
```

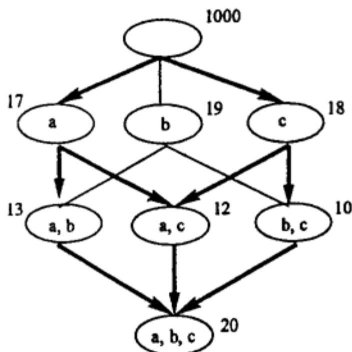

Figure 3.5. Beam search illustration with three features a, b, c, $\eta = 2$.

the non-monotonicity of a measure. Therefore, Approximate Branch & Bound does not aim to speed up, but to find the optimal set in case of a non-monotonic measure. It runs even longer than Branch & Bound.

Table 3.10. Heuristic search: approximate branch & bound - ABAB.

```
ABAB Algorithm
  Input: F - full set, Q - queue, U - measure
         β - bound for some value of U
         δ - allowable deviation from β

  initialize: node = F
              best = {}
   ABAB (node)
    {
      for all children C of node {
        if (C's value of U ≤ β + δ) {
                      /* else, the branch starting with C is pruned*/
            if (C's value < best's value)
                best = C            /* remember the best subset*/
            ABAB (C)
        }
      }
    }
  Output: best                        /* the best set w.r.t. U */
```

3.3.3 *Nondeterministic search*

For both complete and heuristic search strategies, they share one property in common, i.e., they are all *deterministic*. That means no matter how many times one runs a particular algorithm, he can expect that the solution from any subsequent run is always the same as that of the first run. Here we introduce a set of algorithms that share another property - *nondeterministic*. For such an algorithm, one should not expect the same solution from different runs. One of the major motivations for developing this sort of algorithms is to avoid getting stuck in local minima as in heuristic search. Another motivation is to capture the interdependence of features which heuristic search is also incapable of capturing (See Figure 3.4 where the optimal solution (b, c) is missed).

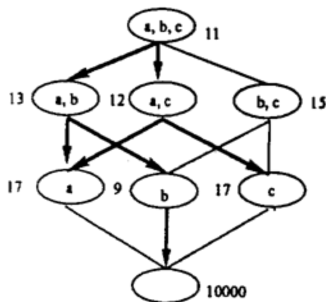

Figure 3.6. Approximate Branch & Bound search illustration with three features a, b, c.

We use one such algorithm shown in Table 3.11 to explain the working of nondeterministic search. **RAND** keeps only the current best subset that satisfies U as well as has the smallest cardinality. As **RAND** continues, it can only produce a better subset. Statement "print S_{best}" reports a better subset found. **RAND** is an any-time algorithm (Boddy and Dean, 1994). It means that one need not wait until the end of **RAND**'s running in order to find a subset. But can it find optimal subsets? If it is allowed a sufficiently long running period and armed with a good random function, yes, it can. This leads us to the question when it should stop. The problem with this scheme is that we don't know if we have found the optimal subset, we only know that if there is a better one than S_{best}, **RAND** will find it. In some cases, one cannot allow a program to run forever, therefore, a stopping criterion can be a number of maximum loops - this guarantees the termination of **RAND**, or the minimum cardinality of S_{best} - obviously this may not assure the stop of **RAND**. Since **RAND** starts with F and ends with S_{best}, it works in a backward fashion in terms of finding the solution. The way in which features are generated is random.

Researchers (Siedlecki and Sklansky, 1988) also proposed to apply Genetic algorithms, and Simulated annealing to feature selection. In a genetic algorithm (Goldberg, 1989), a solution is usually represented by a finite sequence of numbers (such a string is called a *chromosome*, each number is a *gene*). The algorithm manipulates a set of chromosomes (the population), in a manner resembling the mechanism of natural evolution. The chromosomes are allowed to crossover or mutate to produce chromosomes of the next generation. A crossover of two chromosomes produces offspring chromosomes. For instance,

Table 3.11. Random search - RAND.

```
RAND Algorithm
  Input: F - full set
         U - measure
  initialize: S = S_best = {}              /*S - subset set*/
              C_best = #(F)                /*# - cardinality of a set*/
  repeat
     S = RandGen(F)
     C = #(S)
     if C ≤ C_best ∧ S satisfies U
        S_best = S
        C_best = C
        print S_best
  until some stopping criterion is satisfied
  Output: S_best                           /*Best set found so far*/
```

we have two chromosomes (11|000) and (00|111) where | specifies the crossover point, the crossover of these two chromosomes produces two new chromosomes (00|000) and (11|111). If each gene represents a feature, the two new chromosomes represent one empty and one full set. A mutation of a chromosome produces a near identical copy with some components of the chromosome altered. After mutation on the second gene, a chromosome (00|111) becomes (01|111). A fitness function is required to evaluate the fitness of each chromosome. The fittest survives. If the population has n chromosomes, after mutation and crossover, only the fittest n chromosomes will make to the next generation of n chromosomes. A fitness function is, in the context of feature selection, an evaluation measure. The number of generations a genetic algorithm should produce can be pre-determined. When the algorithm converges (the population stays unchanged), solutions are reached.

A simulated annealing algorithm transforms an optimization problem in a problem-specific manner into an annealing problem. Without loss of generality, the objective function (an evaluation measure) is assumed to be minimized. Following an annealing schedule, the temperature is set high in the beginning to allow sufficient active activities (hoping to avoid getting stuck in local minima), then gradually cools down to certain equilibrium representing the best subset. Neural networks plus node pruning (Setiono and Liu, 1997) can also be applied to feature selection. The idea is straightforward: train a multi-layer perceptron as usual following back-propagation, then prune the connections between layers as much as possible without sacrificing the predictive accuracy. The input units

without connections to the hidden units can be removed. The remaining input units are selected features.

Among the four nondeterministic algorithms, RAND and genetic algorithms produce multiple solutions, but simulated annealing and neural networks give single solution. In (Jain and Zongker, 1997) they observe such difference in reviewing these and some other feature selection algorithms. All of the above algorithms rely on some evaluation measures to determine the ranks of features or legitimacy of subsets. With different measures, more variations of feature selection methods are possible. In the next section, we introduce and discuss some representative evaluation measures.

3.4 EVALUATION MEASURES WITH EXAMPLES

We recapitulate that there are three types of evaluation measures, i.e., classic, consistency and accuracy. Classic measures are further divided into information, distance, and dependence measures. For two subsets of features, S_i and S_j, one is preferred to the other based on a measure U of feature-set evaluation. S_i and S_j are indifferent if $U(S_i) = U(S_j)$ and $\#(S_i) = \#(S_j)$ where $\#$ is the cardinality of a set; S_i is preferred to S_j if $U(S_i) = U(S_j)$ but $\#(S_i) < \#(S_j)$, or if $U(S_i) < U(S_j)$ and $\#(S_i) \leq \#(S_j)$. We continue here the discussion about measures in Section 2 of Chapter 2, give details about representative measures in each type to such an extent that a reader can implement them with minimum efforts.

In order to show the working of each chosen measure, we use the data introduced in Chapter 1 and reproduce it here in Table 3.12 for easy reference. For each measure, we give its definition and some details for implementation, and show by example how it works for the data in Table 3.12. Data are translated into numbers where for attribute Hair, blonde = 1, brown = 2, and red = 3; for attribute Height, short = 1, average = 2, and tall = 3; for attribute Weight, light = 1, average = 2, heavy = 3; for attribute Lotion, yes = 1, no = 0; and for class attribute Result, sunburned = 1, and none = 0.

3.4.1 Classic measures

The measures here have existed for quite some time and extensively used in pattern recognition (Ben-Bassat, 1982). Many variations have also been proposed and implemented. As we know, the way of evaluating features is influenced by feature generation scheme and search strategy. Here we offer the very first step of feature evaluation - using one classic measure to choose the best feature. The next step of feature evaluation is about evaluating feature subsets, which is related to feature generation and search strategy. One heuristic method can be: after the best feature is chosen, we find the second best feature by pair-

Table 3.12. An example of feature-based data - revisited.

	Hair	Height	Weight	Lotion	Result
i_1	1	2	1	0	1
i_2	1	3	2	1	0
i_3	2	1	2	1	0
i_4	1	1	2	0	1
i_5	3	2	3	0	1
i_6	2	3	3	0	0
i_7	2	2	3	0	0
i_8	1	1	1	1	0

ing each unchosen feature with the best feature, and select the best pair, then the best triplet, etc. The run time complexity for the heuristic procedure is $O(N^2)$. One should not be surprised if the ranked list of features thus obtained is not the same as the ranked list obtained in the first step, nor the optimal list obtained from complete search.

Before we present the classic measures, some notations are again given for the reader's convenience: $P(c_i)$ is the prior probabilities for all classes i, and $P(\mathbf{x}|c_i)$ is the conditional probabilities of $\mathbf{x}$ given class c_i. By Bayes' theorem, we have

$$P(c_i|\mathbf{x}) = \frac{P(c_i)P(\mathbf{x}|c_i)}{P(\mathbf{x})}, \tag{3.1}$$

$$P(\mathbf{x}) = \sum P(c_i)P(\mathbf{x}|c_i). \tag{3.2}$$

In other words, we only need two groups of probabilities ($P(c_i)$, $P(\mathbf{x}|c_i)$) in order to get $P(c_i|\mathbf{x})$. As an example, we list the priors and class conditional probabilities in Table 3.13.

Information Gain. Shannon's entropy is used here as an example of information gain measure. In Figure 3.7, data D is split by feature X into p partitions $D_1, D_2, ..., D_p$, and there are d classes. The information for D at the root amounts to

$$I(D) = -\sum_{i=1}^{d} P_D(c_i) log_2 P_D(c_i),$$

the information for D_j due to partitioning D at X is

$$I(D_j^X) = -\sum_{i=1}^{d} P_{D_j^X}(c_i) log_2 P_{D_j^X}(c_i),$$

Table 3.13. Priors and class conditional probabilities for the sunburn data.

	Result (Sunburn)	
	No	Yes
P(Result)	5/8	3/8
P(Hair=1|Result)	2/5	2/3
P(Hair=2|Result)	3/5	0
P(Hair=3|Result)	0	1/3
P(Height=1|Result)	2/5	1/3
P(Height=2|Result)	1/5	2/3
P(Height=3|Result)	2/5	0
P(Weight=1|Result)	1/5	1/3
P(Weight=2|Result)	2/5	1/3
P(Weight=3|Result)	2/5	1/3
P(Lotion=0|Result)	2/5	3/3
P(Lotion=1|Result)	3/5	0

and the information gain due to feature X is defined as

$$IG(X) = I(D) - \sum_{j=1}^{p} \frac{|D_j|}{|D|} I(D_j^X),$$

where $|D|$ is the number of instances in D, and $P_D(c_i)$ are priors for data D.

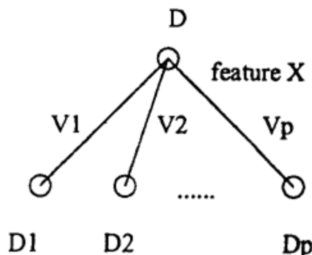

Figure 3.7. Information gain calculation example: Data D is partitioned by feature X into data subsets D_i, $i = 1, 2, ..., p$.

Table 3.14. Ordering features according to their information gains.

```
Information-Gain
  Input: D - the training data set;
         A_i - all features, i = 1, 2, ..., N
  initialize: L = {}                          /*L - empty list*/
  for i = 1 to N
    begin
      calculate IG(A_i);
      insert A_i in L in descending order w.r.t. IG(A_i);
    end
  Output: L                      /*The first A_i in L is the best*/
```

A feature ordering algorithm using information gain is shown in Table 3.14. Its time complexity to obtain the ranked list L is $O(N^2)$ where N is the number of features. However, it is only $O(N)$ if the best feature is sought.

Let's look at feature *Hair* which has three values (1, 2, 3). With this feature, we could partition the data into $D_1^{Hair} = \{i_1(1), i_2(0), i_4(1), i_8(0)\}$, $D_2^{Hair} = \{i_3(0), i_6(0), i_7(0)\}$, and $D_3^{Hair} = \{i_5(1)\}$, where $i_j(k)$ represents instance j with class value k. Since $I(D) = -\frac{5}{8}log_2\frac{5}{8} - \frac{3}{8}log_2\frac{3}{8} = 0.954434$, $I(D_1^{Hair}) = 2 \times (-0.5log_2 0.5) = 1$, $I(D_2^{Hair}) = 0$, and $I(D_3^{Hair}) = 0$, $IG(Hair) = I(D) - \frac{4}{8}I(D^{Hair}) = 0.454434$. This gain happens to be the largest among the four $IG(X)$: $IG(Lotion) = 0.347590$, $IG(Height) = 0.265712$, and $IG(Weight) = 0.015712$. Hence, feature $Hair$ is the first in the ranked list.

Information gain has a tendency to choose features with more distinct values. Information gain ratio was suggested by (Quinlan, 1988) to balance the effect of many values. It is only applied to discrete features; for continuous features, a split point is found with the highest gain or gain ratio among the sorted values in order to split the values into two segments. Thus, information gain can be calculated as usual.

Distance Measures. We can evaluate features using distance measures. Simply replacing IG with DD - directed divergence or V - variance, we can obtain two more algorithms. Since they are very similar to the **Information-Gain** algorithm, we will only show their definitions and calculations. In both measures, we need $P(c_i|\mathbf{x})$ which can be calculated via Equation 3.1, $P(c_i)$ and $P(\mathbf{x}|c_i)$ can be found in Table 3.13.

1. Directed divergence DD is shown in Equation 3.3. The features are ranked as (Hair, Lotion, Height, and Weight) because DD(Hair)= 0.454434, DD(Height)= 0.265712, DD(Weight)= 0.015712, and DD(Lotion)= 0.347590.

$$DD(X_j) = \int [\sum P(c_i|X_j = x) log \frac{P(c_i|X_j = x)}{P(c_i)}] P(X_j = x)dx. \qquad (3.3)$$

2. Variance V is shown in Equation 3.4. The features are ranked as (Lotion, Hair, Height, and Weight) because V(Hair)= 0.059082, V(Height)= 0.054525, V(Weight)= 0.005208, and V(Lotion)= 0.064600.

$$V(X_j) = \int [\sum P(c_i)(P(c_i|X_j = x) - P(c_i))^2] P(X_j = x)dx. \qquad (3.4)$$

Dependence Measures. If we replace information gain with a dependence measure in the algorithm shown in Table 3.14, we obtain another algorithm. As an example, if we adopt Bhattacharyya dependence measure B in Equation 3.5, a ranked list is obtained as (Hair, Lotion, Height, Weight) because B(Hair)= 2.566696, B(Height)= 2.354554, B(Weight)= 2.101802, and B(Lotion)= 2.469646.

$P(x)$ can be obtained following Equation 3.2 and priors and class conditional probabilities are found in Table 3.13.

$$B(X_j) = \sum -log[P(c_i) \int \sqrt{P(X_j = x|c_i)P(X_j = x)} dx] \qquad (3.5)$$

Summary. In the above, we give examples of ranking individual features for the classic measures. They can be extended to evaluating subsets of features, although the number of combinations of features is usually too large as we know. In order to reduce the run time complexity, one way of gradually adding features is to find the best feature, then find among the rest another feature that combines with the selected feature(s) to form a new subset of selected features. More will be elucidated using feature selection methods as examples in Chapter 4.

3.4.2 Consistency measures

We introduce *inconsistency rate* as one of consistency measures. Zero inconsistency means total consistency. Let U be an inconsistency rate over the dataset

given a feature subset S_i. The inconsistency rate is calculated as follows: (1) two instances are considered inconsistent if they are the same except for their class labels (we call these instances as matching instances), for example, for two instances (0 1 1) and (0 1 0), their values of the first two features are the same (0 1), but their class labels are different (1 and 0); (2) the inconsistency count is the number of all the matching instances minus the largest number of instances of different class labels: for example, in n matching instances, c_1 instances belong to label$_1$, c_2 to label$_2$, and c_3 to label$_3$ where $c_1 + c_2 + c_3 = n$. If c_3 is the largest among the three, the inconsistency count is $(n - c_3)$; and (3) the inconsistency rate is the sum of all the inconsistency counts divided by the total number of instances ($\mathcal{N}$). By employing a hashing mechanism, we can compute the inconsistency rate approximately with a time complexity of $O(\mathcal{N})$. Any standard data structure text books should have descriptions on various hashing mechanisms. The interested reader may consult (Kruse et al., 1997).

Here we give the proof outline to show that this inconsistency rate measure is monotonic, i.e., for two subsets S_i and S_j, if $S_i \subset S_j$, then $U(S_i) \geq U(S_j)$. Since $S_i \subset S_j$, the discriminating power of S_i can be no greater than that of S_j. As we know, the discriminating power is reversely proportional to the inconsistency rate. Hence, the inconsistency rate of S_i is greater than or equal to that of S_j, or $U(S_i) \geq U(S_j)$. The monotonicity of the measure can also be analyzed as follows. Consider three simple cases of $S_k (= S_j - S_i)$ without loss of generality: (i) features in S_k are irrelevant, (ii) features in S_k are redundant, and (iii) features in S_k are relevant. (We consider here data without noise.) If features in S_k are irrelevant, based on the definition of irrelevancy, these extra features do not change the inconsistency rate of S_j since S_j is $S_i \cup S_k$, so $U(S_j) = U(S_i)$. Likewise for case (ii) based on the definition of redundancy. If features in S_k are relevant, that means S_i does not have as many relevant features as S_j. Obviously, $U(S_i) \geq U(S_j)$ in the case of $S_i \subset S_j$. It is clear that the above results remain true for cases that S_k contains irrelevant, redundant as well as relevant features.

As for the sunburn example, the inconsistency rate is zero for the following cases: (1) two features (Hair and Lotion); (2) first three features (Hair, Height and Weight); and (3) all super-sets of cases (1) and (2), including the full set of four features. Since we prefer the subset with the minimum number of features, we choose case (1). Intuitively, it also makes more sense than case (2). Why should Height and Weight be linked to sunburned? This is a hindsight, of course, but it is also an effective way to verify if the results are sensible using our knowledge. Case (1) can be obtained by a sequential forward heuristic algorithm, and case (2) can be found by a sequential backward heuristic algorithm.

Since the condition for Branch & Bound to work optimally is that its evaluation measure be monotonic and this inconsistency rate measure is monotonic, we can expect to find an optimal feature subset by applying the inconsistency rate measure to Branch & Bound. Consistency measures are only suitable for discrete features. For continuous features, they have to be discretized before this kind of measure can be applied (Catlett, 1991, Kerber, 1992).

3.4.3 Accuracy measures

In theory, any classifier is qualified to provide predictive accuracy as an evaluation measure. However, in practice, we are usually constrained by many factors. Among many, we list three factors: (1) the choice of a classifier - many classification algorithms are extant for different needs (Mitch, 1997), we should always keep it in mind that feature selection is performed to improve a classifier; (2) the speed of learning - how fast can a classifier be induced plays an important role in practice, some applications can wait, but others cannot; and (3) the task at hand - generalization from the data is a goal, we do have tasks that may also emphasize things like explicitness, simplicity, or comprehensibility. We may choose one classifier which suits the task at hand best. However, due to the time constraint, we may exclude those time consuming learning algorithms such as neural networks, and genetic algorithms. Here we simplify the issue and assume no specific requirement on a classifier.

A common way of applying accuracy measure is Sequential Backward Search. The idea is to see if the classifier can maintain its accuracy by removing one feature at a time until there is only one feature or until accuracy deteriorates to an intolerable level. In Table 3.15, we present such an algorithm using the number of features as the stopping criterion. This algorithm's run time complexity is $O(N^2)$ since it removes one feature at a time until only one feature remains. However, we need to bear in mind that constructing a classifier, as in `induce()`, often requires a similar order of complexity. Its time complexity varies with different classification algorithms, so it is not given here. Function **`reorganize`** arranges the remaining features in order for easy indexing.

In the sunburn example, we can use a Naive Bayesian Classifier (NBC) in `induce()`. With feature set F, NBC's accuracy is 100%. By removing one of four features in turn, we obtain the following according to their accuracy rates ($ER = 1-$ accuracy):

$S_1 = F - \{\text{Hair}\}$ 87.5%
$S_2 = F - \{\text{Height}\}$ 100%
$S_3 = F - \{\text{Weight}\}$ 100%
$S_4 = F - \{\text{Lotion}\}$ 75%

Table 3.15. Ordering features according to accuracy.

```
Accuracy
  Input: D - the training data set;
         F - the full set of features;
         A_i - feature i, i = 1,2,...,N
  initialize: S = F                          /*S - full feature set*/
              L = {}                         /*L - empty list */
  ER = induce(D,S)                 /*ER - error rate with set S*/
  repeat
    min = ER;
    for i = 1 to N
      begin
        S_i = S - A_i
        ER_i = induce(D,S_i)
        if min > ER_i
          min = ER_i
          index = i
      end
    N = N - 1
    remove A_index from S and append it to L
    reorganize S from 1 to N
  until N = 1
  append A_1 to L           /*Features in L are in original indexes*/
  Output: reversed L              /*The first A_i in L is the best*/
```

So, we can remove features *Height* and *Weight* sequentially (S_5 = {Hair, Lotion}) and repeat the removal of one feature procedure, we have

$S_6 = S_5 - \{\text{Hair}\}$ 75%

$S_7 = S_5 - \{\text{Lotion}\}$ 75%

Based on the results above, we have a ranked list {Lotion, Hair, Weight, Height }.

3.5 CONCLUSION

In this chapter, we study feature selection in three individual aspects: search direction (how features are generated), search strategy, and evaluation measure. The isolation of one aspect from the others allows us to focus on the specific problems each aspect experiences and on the possibilities each aspect can have. When we discuss each individual aspect, we cannot help sense the interweaving

effects of the others. A particular method of feature selection is basically a combination of some possibilities of every aspect. In the beginning of the book, we started our analysis of perspectives of feature selection in a top-down approach; after studying each aspect of feature selection here, we are ready to go up in a bottom-up manner. Combining aspects with different possibilities, we will be able to create or reproduce many effective feature selection methods.

References

Almuallim, H. and Dietterich, T. (1994). Learning boolean concepts in the presence of many irrelevant features. *Artificial Intelligence*, 69(1-2):279–305.

Ben-Bassat, M. (1982). Pattern recognition and reduction of dimensionality. In Krishnaiah, P. R. and Kanal, L. N., editors, *Handbook of statistics-II*, pages 773–791. North Holland.

Boddy, M. and Dean, T. (1994). Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–285.

Catlett, J. (1991). On changing continuous attributes into ordered discrete attributes. In *European Working Session on Learning*.

Cover, T. (1974). The best two independent measurements are not the two best. *IEEE Trans. Systems, Man and Cybernictics*, 4:116–117.

Dash, M. and Liu, H. (1997). Feature selection methods for classifications. *Intelligent Data Analysis: An International Journal*, 1(3).

Dietterich, T. (1997). Machine learning research: Four current directions. *AI Magazine*, pages 97–136.

Domingos, P. (1997). Why does bagging work? a Bayesian account and its implications. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, pages 155 – 158. AAAI Press.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Company, Inc.

Jain, A. and Zongker, D. (1997). Feature selection: Evaluation, application, and small sample performance. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 19(2):153–158.

John, G., Kohavi, R., and Pfleger, K. (1994). Irrelevant feature and the subset selection problem. In *Machine Learning: Proceedings of the Eleventh International Conference*, pages 121–129. Morgan Kaufmann Publisher.

Kerber, R. (1992). ChiMerge: Discretization of numeric attributes. In *AAAI-92, Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 123–128. AAAI Press/The MIT Press.

Kruse, R., Tondo, C., and Leung, B. (1997). Data structures & program design in C. International Edition.

Liu, H. and Wen, W. (1993). Concept learning through feature selection. In *Proceedings of the First Australian and New Zealand Conference on Intelligent Information Systems*, pages 293–297.

Mitch, T. (1997). *Machine Learning*. McGraw-Hill.

Narendra, P. and Fukunaga, K. (1977). A branch and bound algorithm for feature subset selection. *IEEE Trans. on Computer*, C-26(9):917–922.

Press, W., Teukolsky, S., Vetterling, W., and Flannery, B. (1992). *Numerical Recipes in C*. Cambridge University Press, Cambridge.

Quinlan, J. (1988). Decision trees and multi-values attributes. In J.E., H., Michie, D., and J., R., editors, *Machine Intelligence*, volume 11. Oxford University Press.

Quinlan, J. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

Setiono, R. and Liu, H. (1997). Neural network feature selectors. *IEEE Trans. on Neural Networks*, 8(3):654–662.

Siedlecki, W. and Sklansky, J. (1988). On automatic feature selection. *International Journal of Pattern Recognition and Artificial Intelligence*, 2:197–220.