Chapter 17
Case Study: Job Scheduling

High-performance computing (HPC) environments are used by many technology and research organizations to facilitate large-scale computations. An HPC environment usually consists of numerous "compute nodes" networked to handle computations required by the users. These can be structured in different ways, such as a network of many computers each with fewer processors, or a network of fewer computers with many processors. The amount of memory on each may vary from environment to environment and often is built based on a balance between available funds and the nature of the types of computations required by an institution.

A specific unit of computations (generally called a *job* here) can be launched by users for execution in the HPC environment. The issue is that, in many cases, a large number of programs are executed simultaneously and the environment has to manage these jobs in a way that returns the job results in the most efficient manner. This may be a complicated task. For example:

- There are times where the existing jobs outnumber the capacity of the environment. In these cases, some jobs will be required to stay pending prior to launch until the appropriate resources are available.
- The frequency of submissions by users may vary. If one user submits a large number of jobs at once, it may be undesirable for the system to let a single user consume the majority of resources at the expense of the other users.
- All compute jobs may not be treated equally. Higher priority projects may require more privileged access to the hardware resources. For example, jobs at a pharmaceutical company needed to support a time-sensitive regulatory submission may need to have a higher priority than jobs related to research-oriented tasks.

A common solution to this problem is to use a *job scheduler*—a software that prioritizes jobs for submissions, manages the computational resources, and initiates submitted jobs to maximize efficiency. The scheduler can enact a queuing system based on several factors such as resource requirements for

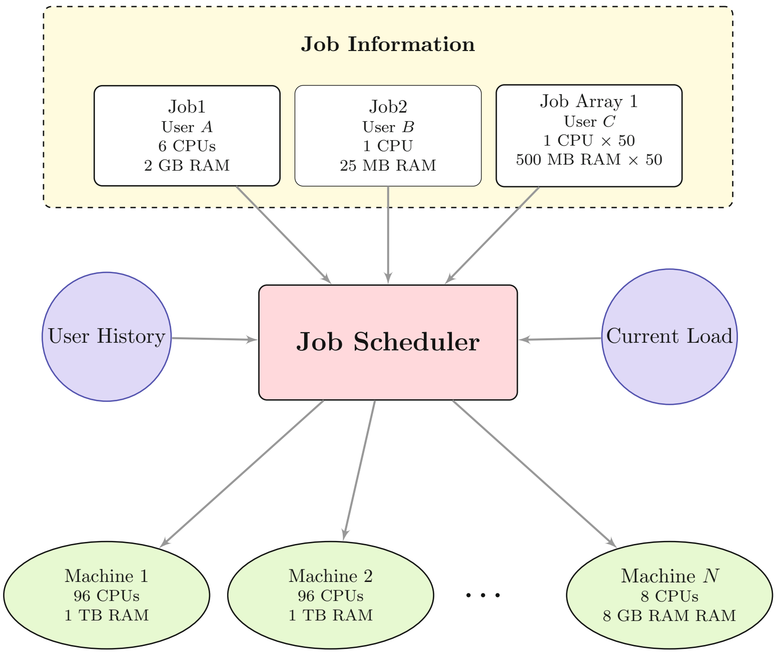


Fig. 17.1: A hypothetical example of a job scheduler in a high-performance computing environment

a job (e.g., number of processors required, memory requirements), project priority, and the current load of the environment. The scheduler may also consider the submission history of specific users when delegating jobs to computational resources.

Figure 17.1 shows a schematic of such a system. The top shows three hypothetical jobs that are submitted at the same time. The first two have a single set of computations each but have different requirements. For example, the first job could be a scientific computation that spreads the computations across multiple processors on the same physical machine. The second job may be a short database query that a scientist makes interactively from a web site. This job is expected to run quickly with low overhead. The third case represents a cohesive set of computations that are launched at once but correspond to a set of independent calculations (such as a simulation) that use the same program. This *job array* launches 50 independent calculations that can be run on different machines but should be managed and supervised as a single entity. Again the goal of the scheduler is to allocate these jobs to the system in a manner that maximizes efficiency.

The efficiency of the scheduler can be significantly affected by the amount and quality of job information that is known at the time of submission. This allows the scheduler to place the jobs on suitable machines at the appropriate time. For computations that are repeated a large number of times, it is straightforward to record information on existing jobs then make resource predictions on new jobs. For example, it may be helpful to predict the execution time or maximum memory requirements for a job at the time of submission. The requirements for such a prediction are that it be quick and accurate. However, the accuracy requirements should take into account the fact that prediction errors do not likely have the same impact on the environment's performance. For example, if the predicted amount of memory needed is severely underestimated, this could cause physical machine resources to be overburdened and may drastically affect all the jobs on that machine. The cost of this type of error should be high in order for the scheduler to avoid this problem. The converse problem is not true: overestimation of memory needs may cause abnormally low utilization of a hardware resource but will not critically affect existing jobs on the system. A cost should be assigned to this error, but likely not as high as the other type of error.

As an example, Pfizer runs a large number of jobs in its HPC environment. One class of jobs for characterizing compounds is run regularly, some of which can be computationally burdensome. Over a period of time, resource utilization information was logged about this class of jobs. Additionally, there are several task characteristics that can be collected at the time of job launch:

- Protocol: Several different analytical techniques, called *protocols*, can be used to calculate results for each compound. The protocols are coded as letters `A` through `O`; protocol `J` was used the most (22.8 % of all the jobs) while protocol `K` was rarely invoked (0.1 %).
- Number of compounds: The number of compounds processed by the job. This number varied wildly across the data.
- Number of input fields: Each task can process a number of different input fields in the data, depending on what the scientist would like to analyze. This predictor is also right skewed.
- Number of iterations: Each protocol can run for a pre-specified number of iterations. The default is 20 iterations, which occurs most frequently in these data.
- Pending job count: A count of how many jobs were pending at the time launch was recorded. This is meant to measure the workload of the environment at the time of the launch. All other things being equal, the number of pending jobs should not directly affect execution time of the jobs but may capture the amount of resources being used at the time. For these data, most jobs were launched at a time when there were available resources, so no jobs were pending. However, a minority of requests were made when the hardware was in high demand (thousands of jobs were already pending).
- Time of day: The time of day (Eastern Standard Time) that the job was launched (0 to 24). The distribution is multimodal, which reflects users coming online in three different time zones on two continents.
- Day of the week: The day of the week when the job was launched.

Execution time was recorded for each job. Time spent in pending or suspended states were not counted in these values. While the outcome is continuous in nature, the scheduler required a qualitative representation for this information. Jobs were required to be classified as either very fast (1 m or less), fast (1–5 m), moderate (5–30 m), or long (greater than 30 m). Most of the jobs fall into either the very fast category (51.1%) or the fast category (31.1%) while only 11.9% were classified as moderate and 6% were long.

The goal of this experiment is to predict the class of the jobs using the information given in Table 17.1. Clearly, the types of errors are not equal and there should be a greater penalty for classifying jobs as very short that are in fact long. Also, since the prediction equation will need to be implemented in software and quickly computed, models with simpler prediction equations are preferred.

Over the course of two years, changes were made to some of the hardware in the HPC environment. As a consequence, the same job run on two different vintages of hardware may produce different execution times. This has the effect of inflating the inherent variation in the execution times and can lead to potential mislabeling of the observed classes. While there is no way to handle this in the analysis, any model for classifying execution times should be revisited over time (on the assumption that new hardware will decrease execution time).

Before delving into model building, it is important to investigate the data. Prior to modeling, one would expect that the main drivers of the execution time would be the number of compounds, the number of tasks, and, which protocol is being executed. As an initial investigation, Fig. 17.2 shows the relationship between the protocols and the execution time classes using a *mosaic plot*. Here the widths of the boxes are indicative of the number of jobs run for the protocol (e.g., protocol `J` was run the most and `K` the least). To predict the slow jobs, one can see that only a few protocols produce long execution times. Others, such as protocol `D`, are more likely to be executed very quickly. Given these relationships, we might expect the protocol information to be potentially important to the model.

Figure 17.3 shows another visualization of the data using a *table plot*. Here, the data are ordered by the class then binned into 100 slices. Within each slice, the average value of the numeric predictors is determined. Similarly, the frequency distribution of the categorical predictors is determined. The results for each predictor are shown in columns so that the modeler can have a better understanding of how each predictor relates to the outcome. In the figure, we can see that:

- The jobs associated with a large number of compounds tend to be either large or moderate in execution time.
- Many of the jobs of moderate length were submitted when the number of pending jobs was very high. However, this trend does not reproduce itself in the very long jobs. Because of this, it would be important to

Table 17.1: Predictors for the job scheduler data

```
                    7 Variables      4331  Observations
```

```
Protocol
    n missing unique
 4331       0     14

            A   C   D  E   F   G   H   I   J K   L   M   N   O
Frequency 94 160 149 96 170 155 321 381 989 6 242 451 536 581
%          2   4   3  2   4   4   7   9  23 0   6  10  12  13
```

```
Compounds
```

```
    n missing unique  Mean 0.05 0.10 0.25 0.50 0.75 0.90 0.95
 4331       0    858 497.7   27   37   98  226  448  967 2512

lowest :    20    21    22    23    24
highest: 14087 14090 14091 14097 14103
```

```
InputFields
```

```
    n missing unique Mean 0.05 0.10 0.25 0.50 0.75  0.90  0.95
 4331       0   1730 1537   26   48  134  426  991  4165  7594

lowest :    10    11    12    13    14
highest: 36021 45420 45628 55920 56671
```

```
Iterations
    n missing unique  Mean 0.05 0.10 0.25 0.50 0.75 0.90 0.95
 4331       0     11 29.24   10   20   20   20   20   50  100

           10 11 15   20 30 40  50 100 125 150 200
Frequency 272  9  2 3568  3  7 153 188   1   2 126
%           6  0  0   82  0  0   4   4   0   0   3
```

```
NumPending
```

```
    n missing unique  Mean 0.05 0.10 0.25 0.50 0.75 0.90  0.95
 4331       0    303 53.39  0.0  0.0  0.0  0.0  0.0 33.0 145.5

lowest :    0    1    2    3    4, highest: 3822 3870 3878 5547 5605
```

```
Hour
```

```
    n missing unique  Mean  0.05  0.10   0.25   0.50   0.75   0.90   0.95
 4331       0    924 13.73 7.025 9.333 10.900 14.017 16.600 18.250 19.658

lowest :  0.01667  0.03333  0.08333  0.10000  0.11667
highest: 23.20000 23.21667 23.35000 23.80000 23.98333
```

```
Day
    n missing unique
 4331       0      7

          Mon Tue Wed Thu Fri Sat Sun
Frequency 692 900 903 720 923  32 161
%          16  21  21  17  21   1   4
```

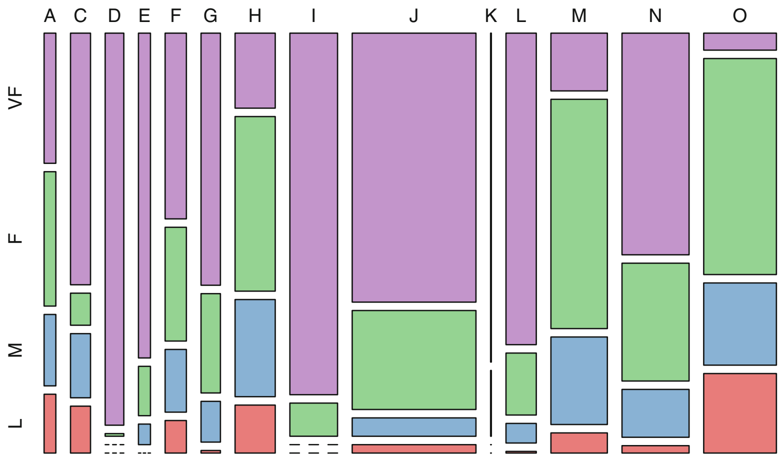


Fig. 17.2: A mosaic plot of the class frequencies for each protocol. The width of the boxes is determined by the number of jobs observed in the data set

prospectively validate this observation to make sure that it is not a fluke of this particular data set.

- When the number of iterations is large, the job tends to go long.

One shortcoming of this particular visualization is that it obscures the relationships between predictors. Correlation plots and scatter plot matrices are effective methods for finding these types of relationships.

Additionally, Fig. 17.4 shows scatter plots for the number of compounds versus the number of input fields by protocol. In these plots, the jobs are colored by class. For some cases, such as protocols `A`, `C`, `D`, `H`, `I`, and `K`, the number of compounds and fields appears to provide information to differentiate the classes. However, these relationships are class specific; the patterns for protocols `I` and `K` are different. Another interesting aspect of this analysis is that the correlation pattern between the number of compounds versus the number of input fields is protocol-specific. For example, there is very little correlation between the two predictors for some protocols (e.g., `J`, `O`) and a strong correlation in others (such as `D`, `E`, and `H`). These patterns might be important and are more likely to be discovered when the modeler looks at the actual data points.

17.1 Data Splitting and Model Strategy

There are 4331 samples available; 80% will be used for training the algorithms while the remainder will be used to evaluate the final candidate models. The data were split using stratified random sampling to preserve the class

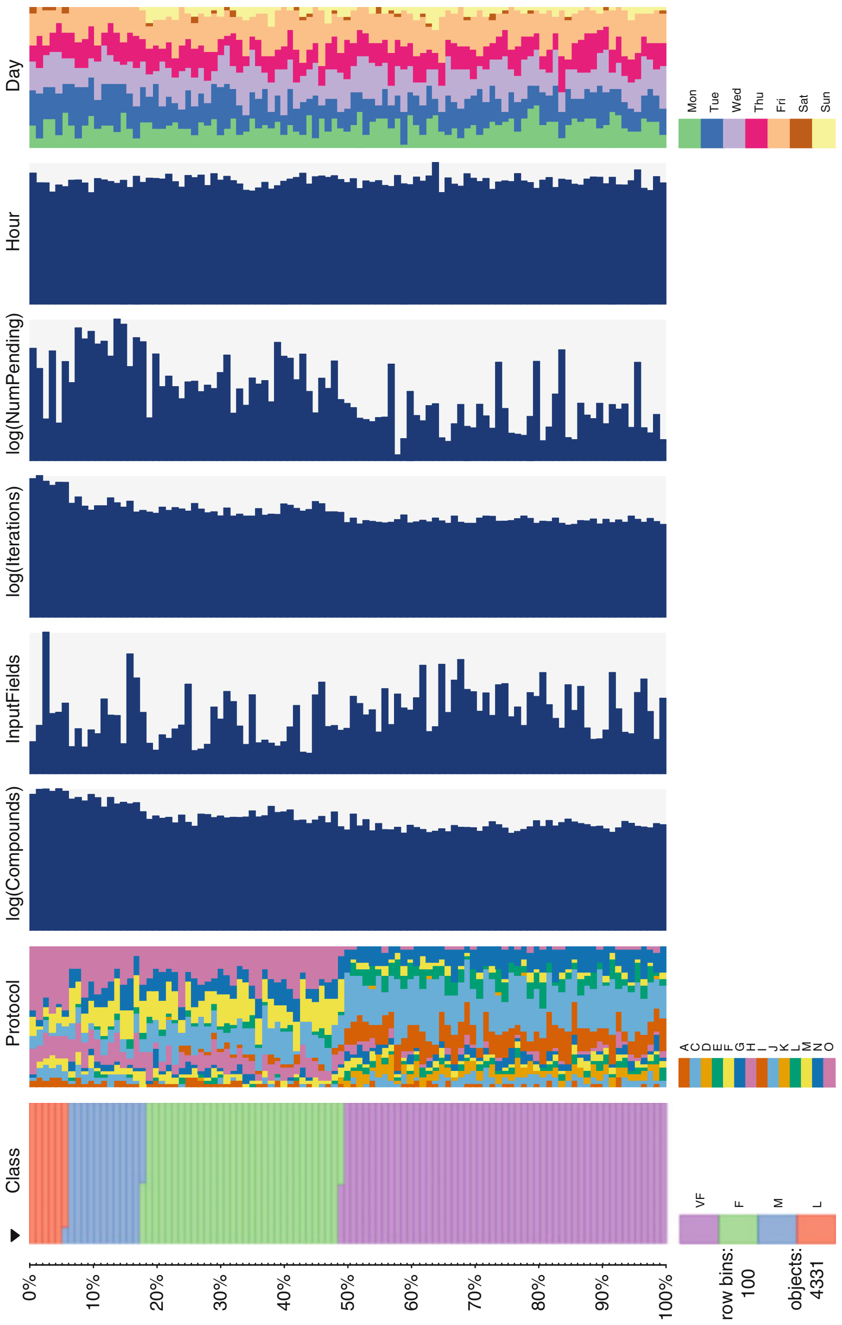


Fig. 17.3: A table plot of the data

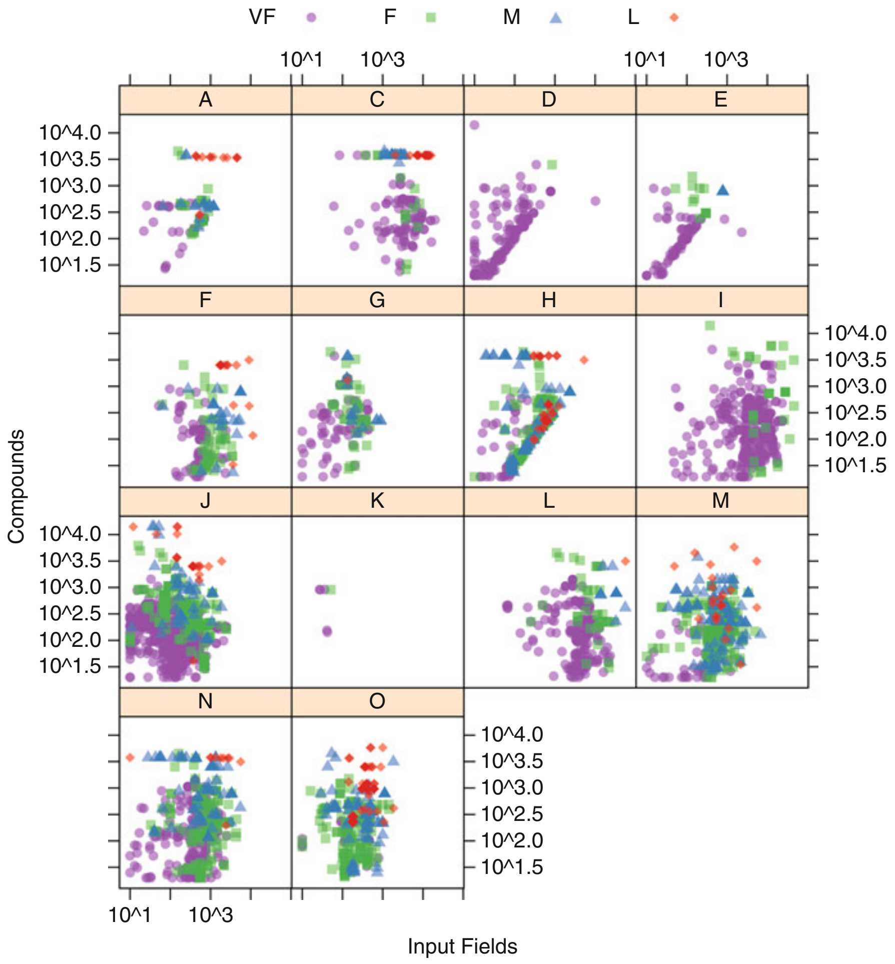


Fig. 17.4: A scatter plot of the number of compounds versus the number of fields for each protocol. The points are colored by their execution time class

distribution of the outcome. Five repeats of 10-fold cross–validation were used to tune the models.

Rather than creating models that maximize the overall accuracy or Kappa statistics, a custom cost function is used to give higher weight to errors where long and medium jobs are misclassified as fast or very fast. Table 17.2 shows how the costs associated with each type of error affect the overall measure of performance. The cost is heavily weighted so that long jobs will not be submitted to queues (or hardware) that are designed for small, quick jobs. Moderate-length jobs are also penalized for being misclassified as more efficient jobs.

Table 17.2: Cost structure used to optimize the model

	Observed Class			
	VF	F	M	L
VF	0	1	5	10
F	1	0	5	5
M	1	1	0	1
L	1	1	1	0

Longer jobs that are misclassified as fast are penalized more in this criterion

The best possible approach to model training is when this cost function is used to estimate the model parameters (see Sect. 16.8). A few tree-based models allow for this, but most of the models considered here do not.

A series of models were fit to the training set. The tuning parameter combination associated with the smallest average cost value was chosen for the final model and used in conjunction with the entire training set. The following models were investigated:

- Linear discriminant analysis: This model was created using the standard set of equations as well as with the penalized version that conducts feature selection during model training. Predictor subset sizes ranging from 2 to 112 were investigated in conjunction with several values of the ridge penalty: 0, 0.01, 0.1, 1 and 10.
- Partial least squares discriminant analysis: The PLS model was fit with the number of components ranging from 1 to 91.
- Neural networks: Models were fit with hidden units ranging from 1 to 19 and 5 weight decay values: 0, 0.001, 0.01, 0.1, and 0.5.
- Flexible discriminant analysis: First-degree MARS hinge functions were used and the number of retained terms was varied from 2 to 23.
- Support vector machines (SVMs): Two different models were fit with the radial basis function. One using equal weights per class and another where the moderate jobs were given a fivefold weight and long jobs were upweighted tenfold. In each case, the analytical calculations for estimating the RBF kernel function were used in conjunction with cost values ranging from 2^{-2} to 2^{12} in the log scale.
- Single CART trees: Similarly, the CART models were fit with equal costs per class and another where the costs mimic those in Table 17.2. In each case, the model was tuned over 20 values of the complexity parameter.
- Bagged CART trees: These models used 50 bagged CART trees with and without incorporating the cost structure.
- Random forests: The model used 2,000 trees in the forest and was tuned over 6 values of the tuning parameter.

- C5.0: This model was evaluated with tree- and rule-based models, with and without winnowing, with single models and with up to 100 iterations of boosting. An alternative version of this model was fit that utilized the cost structure shown in Table 17.2.

For this application, there is a bias towards models which are conducive to fast predictions. If the prediction equation(s) for the model can be easily encoded in software (such as a simple tree or neural network) or can be executed from a simple command-line interface (such as C5.0), the model is preferred over others.

17.2 Results

The model results are shown in Fig. 17.5 where box plots of the resampling estimates of the mean cost value are shown. The linear models, such as LDA and PLS, did not do well here. Feature selection did not help the linear discriminant model, but this may be due to that model's inability to handle nonlinear class boundaries. FDA also showed poor performance in terms of cost.

There is a cluster of models with average costs that are likely to be equivalent, mostly SVMs and the various tree ensemble methods. Using costs/weights had significant positive effects on the single CART tree and SVMs. Figure 17.6 shows the resampling profiles for these two models in

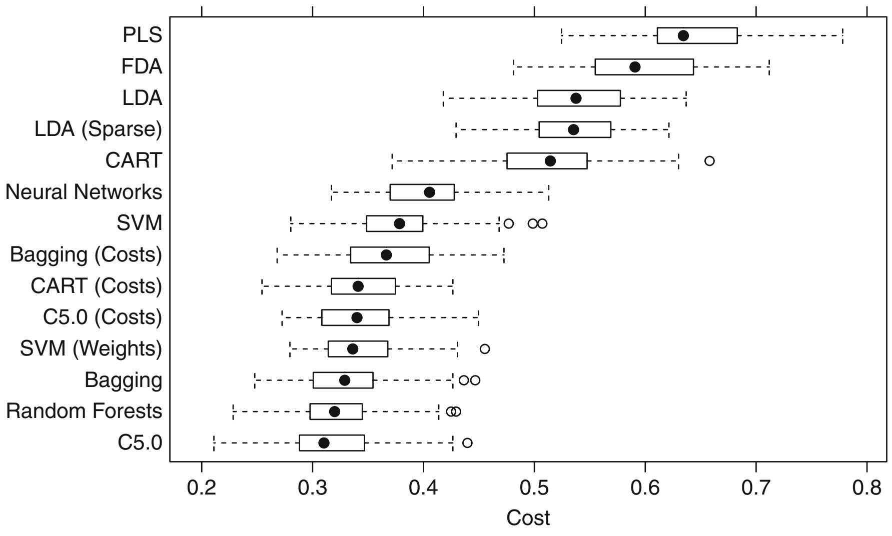
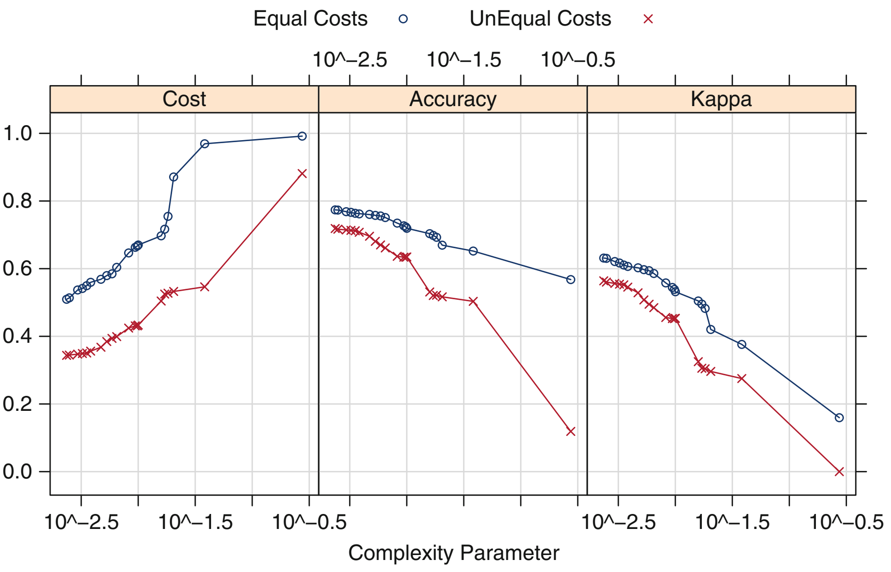


Fig. 17.5: Average cost resampling profiles for various models



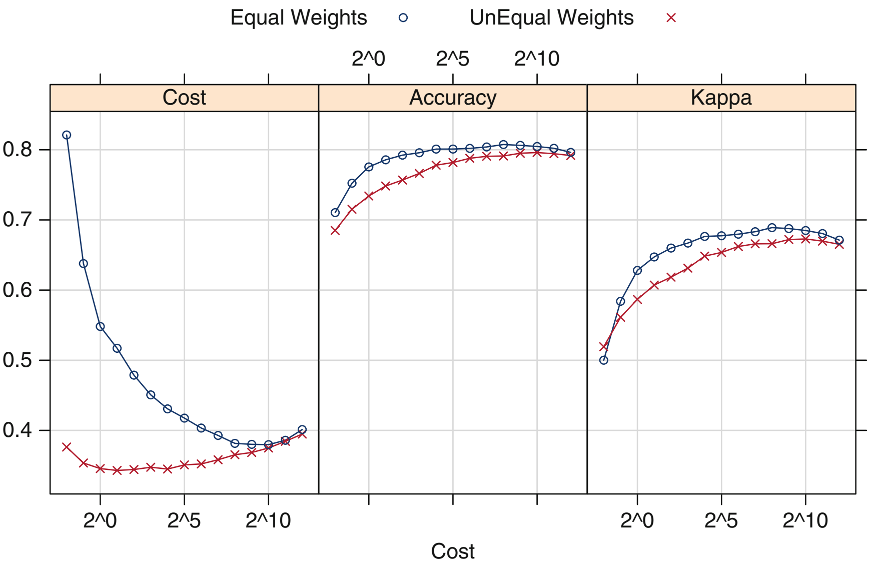


Fig. 17.6: The effect of incorporating the cost structure into the CART training process (*top*) and weights for support vector machines (*bottom*)

terms of their estimates of the cost, overall accuracy and Kappa statistic. The CART model results show that using the cost has a negative effect on accuracy and Kappa but naturally improved the cost estimates. No matter the metric, the tuning process would have picked the same CART model

Table 17.3: Resampled confusion matrices for the random forest and cost-sensitive CART models

	Cost-sensitive CART				Random forests			
	VF	F	M	L	VF	F	M	L
VF	157.0	24.6	2.0	0.2	164.7	18.0	1.3	0.2
F	10.3	43.2	3.1	0.2	11.9	83.7	11.5	1.8
M	9.6	38.3	34.5	5.8	0.2	5.5	27.4	1.8
L	0.0	1.7	1.6	14.6	0.0	0.6	0.9	17.0

Each value is the average number of jobs that occurred in that cell across the 50 holdout data sets. The columns are the true job classes

for final training. The support vector machine (SVM) results are somewhat different. Using class weights also had a slight negative effect on accuracy and Kappa, but the improvement in the estimated cost was significant. Also, the unweighted model would be optimized with a much higher SVM cost parameter (and thus more complex models) than the weighted version.

Oddly, the `C5.0` model without costs did very well, but adding the cost structure to the tree-building process *increased* the estimated cost of the model. Also, bagging CART trees with costs had a small negative effect on the model

Trees clearly did well for these data, as did support vector machines and neural networks. Is there much of a difference between the top models? One approach to examining these results is to look at a confusion matrix generated across the resamples. Recall that for resampling, there were 50 hold–out sets that contained, on average, about 347 jobs. For each one of these hold–out sets, a confusion matrix was calculated and the average confusion matrix was calculated by averaging the cell frequencies.

Table 17.3 shows two such tables for the random forests model and the cost-sensitive CART model. For random forest, the average number of long jobs that were misclassified as very fast was 0.2 while the same value for the classification tree was 0.24. The CART tree shows very poor accuracy for the fast jobs compared to the random forest model. However, the opposite is true for moderately long jobs; random forest misclassified 72.56% of those jobs (on average), compared to 65.52% for the single tree. For long jobs, the single tree has a higher error rate than the ensemble method. How do these two models compare using the test set? The test set cost for random forest was 0.316 while the single classification trees had a average cost of 0.37. Table 17.4 shows the confusion matrices for the two models. The trends in the test set are very similar to the resampled estimates. The single tree does worse with fast and long jobs while random forests have issues predicting the moderately

Table 17.4: Test set confusion matrices for the random forest and cost-sensitive CART models

	Cost-sensitive CART				Random forests			
	VF	F	M	L	VF	F	M	L
VF	383	61	5	1	414	45	3	0
F	32	106	7	2	28	206	27	5
M	26	99	87	15	0	18	71	6
L	1	3	3	33	0	0	1	40

The columns are the true job classes

long jobs. In summary, the overall differences between the two models are not large.

17.3 Computing

The data are contained in the AppliedPredictiveModeling package. After loading the data, a training and test set were created:

```
> library(AppliedPredictiveModeling)
> data(schedulingData)
> set.seed(1104)
> inTrain <- createDataPartition(schedulingData$Class,
+                                p = .8,
+                                list = FALSE)
> schedulingData$NumPending <- schedulingData$NumPending + 1
> trainData <- schedulingData[ inTrain,]
> testData  <- schedulingData[-inTrain,]
```

Since the costs defined in Table 17.2 will be used to judge the models, functions were written to estimate this value from a set of observed and predicted classes:

```
> cost <- function(pred, obs)
+   {
+     isNA <- is.na(pred)
+     if(!all(isNA))
+       {
+         pred <- pred[!isNA]
+         obs <- obs[!isNA]
+         cost <- ifelse(pred == obs, 0, 1)
+         if(any(pred == "VF" & obs == "L"))
+           cost[pred == "L" & obs == "VF"] <- 10
+         if(any(pred == "F" & obs == "L"))
+           cost[pred == "F" & obs == "L"] <- 5
+         if(any(pred == "F" & obs == "M"))
+           cost[pred == "F" & obs == "M"] <- 5
+         if(any(pred == "VF" & obs == "M"))
+           cost[pred == "VF" & obs == "M"] <- 5
+         out <- mean(cost)
+       } else out <- NA
+     out
+   }
> costSummary <- function (data, lev = NULL, model = NULL)
+ {
+     if (is.character(data$obs))  data$obs <- factor(data$obs,
        levels = lev)
+     c(postResample(data[, "pred"], data[, "obs"]),
+       Cost = cost(data[, "pred"], data[, "obs"]))
+ }
```

The latter function is used in the control object for future computations:

```
> ctrl <- trainControl(method = "repeatedcv", repeats = 5,
+                      summaryFunction = costSummary)
```

For the cost-sensitive tree models, a matrix representation of the costs was also created:

```
> costMatrix <- ifelse(diag(4) == 1, 0, 1)
> costMatrix[1,4] <- 10
> costMatrix[1,3] <- 5
> costMatrix[2,4] <- 5
> costMatrix[2,3] <- 5
> rownames(costMatrix) <- levels(trainData$Class)
> colnames(costMatrix) <- levels(trainData$Class)
> costMatrix
```

The tree-based methods did not use independent categories, but the other models require that the categorical predictors (e.g., protocol) are decomposed into dummy variables. A model formula was created that log transforms several of the predictors (given the skewness demonstrated in Table 17.1):

```
> modForm <- as.formula(Class ~ Protocol + log10(Compounds) +
+                       log10(InputFields)+ log10(Iterations) +
+                       log10(NumPending) + Hour + Day)
```

The specifics of the models fit to the data can be found in the `Chapter` directory of the AppliedPredictiveModeling package and follow similar syntax to the code shown in previous chapters. However, the cost-sensitive and weighted model function calls are

```
> ## Cost-Sensitive CART
> set.seed(857)
> rpFitCost <- train(x = trainData[, predictors],
+                    y = trainData$Class,
+                    method = "rpart",
+                    metric = "Cost",
+                    maximize = FALSE,
+                     tuneLength = 20,
+                     ## rpart structures the cost matrix so that
+                     ## the true classes are in rows, so we
+                     ## transpose the cost matrix
+                     parms =list(loss = t(costMatrix)),
+                     trControl = ctrl)

> ## Cost- Sensitive C5.0
> set.seed(857)
> c50Cost <- train(x = trainData[, predictors],
+                  y = trainData$Class,
+                  method = "C5.0",
+                  metric = "Cost",
+                  maximize = FALSE,
+                  costs = costMatrix,
+                  tuneGrid = expand.grid(.trials = c(1, (1:10)*10),
+                                         .model = "tree",
+                                         .winnow = c(TRUE, FALSE)),
+                  trControl = ctrl)

> ## Cost-Sensitive bagged trees
> rpCost <- function(x, y)
+   {
+     costMatrix <- ifelse(diag(4) == 1, 0, 1)
+     costMatrix[4, 1] <- 10
+     costMatrix[3, 1] <- 5
+     costMatrix[4, 2] <- 5
+     costMatrix[3, 2] <- 5
+     library(rpart)
+     tmp <- x
+     tmp$y <- y
+     rpart(y~.,
+           data = tmp,
+           control = rpart.control(cp = 0),
+           parms = list(loss = costMatrix))
+   }
> rpPredict <- function(object, x) predict(object, x)
> rpAgg <- function (x, type = "class")
+ {
+   pooled <- x[[1]] * NA
+   n <- nrow(pooled)
+   classes <- colnames(pooled)
+   for (i in 1:ncol(pooled))
+     {
+       tmp <- lapply(x, function(y, col) y[, col], col = i)
+       tmp <- do.call("rbind", tmp)
+       pooled[, i] <- apply(tmp, 2, median)
+     }
+   pooled <- apply(pooled, 1, function(x) x/sum(x))
+   if (n != nrow(pooled)) pooled <- t(pooled)
+   out <- factor(classes[apply(pooled, 1, which.max)],
+                  levels = classes)
+   out
+ }
> set.seed(857)
> rpCostBag <- train(trainData[, predictors],
+                    trainData$Class,
+                    "bag",
+                    B = 50,
+                    bagControl = bagControl(fit = rpCost,
+                                            predict = rpPredict,
+                                            aggregate = rpAgg,
+                                            downSample = FALSE),
+                    trControl = ctrl)
>


> ## Weighted SVM
> set.seed(857)
> svmRFitCost <- train(modForm, data = trainData,
+                      method = "svmRadial",
+                      metric = "Cost",
+                      maximize = FALSE,
+                      preProc = c("center", "scale"),
+                      class.weights = c(VF = 1, F = 1,
+                                        M = 5, L = 10),
+                      tuneLength = 15,
+                      trControl = ctrl)
```

The resampled versions of the confusion matrices were computed using the `confusionMatrix` function on the objects produced by the `train` function, such as

```
> confusionMatrix(rpFitCost, norm = "none")
  Cross-Validated (10-fold, repeated 5 times) Confusion Matrix

  (entries are un-normalized counts)

            Reference
  Prediction    VF     F     M     L
          VF 157.0  24.6   2.0   0.2
          F   10.3  43.2   3.1   0.2
          M    9.6  38.3  34.5   5.8
          L    0.0   1.7   1.6  14.6
```

The `norm` argument determines how the raw counts from each resample should be normalized. A value of `"none"` results in the average counts in each cell of the table. Using `norm = "overall"` first divides the cell entries by the total number of data points in the table, then averages these percentages.