A Comparison of Learning Techniques in Determining the Success of a Professional Baseball Team.

Vadim von Brzeski
Department of Computer Science
University of California, Santa Cruz

March 12, 2005

Abstract

The goals of this project were twofold : one, to learn the best classifier model for discriminating between winning and losing baseball teams, and two, to refine this model via feature selection to determine the most efficient (smallest number of features) model with the same predictive power as the full model. The candidate learning methods were minimum-squared-error (MSE) classification, boosted decision tree stumps, and support vector machines. The MSE method, using a perceptron-like prediction rule, performed the best, yielding a 8.90% error rate on held out test data, and using only 9 of the 36 features.

1 Introduction

Our classification problem is as follows: given a set of statistics (features) of a professional baseball team in a given year, was the team a "winning team" (+1) or a "losing team" (-1) that year, where "winning team" means that the team's winning percentage for that year was greater than 50%, and "losing team" means that the team's winning percentage for that year was 50% or less. In each year, each team has an associated vector of 36 summary statistics, which we call features, such as the number of runs scored by the team that year, the number of strike-outs by the team that year, etc. (see Appendix A for a complete list of the 36 features and their definitions). This set of features does not include the number of games won or lost, since we are trying to learn whether this is greater or less than 50%.

Our methodology is composed of two phases. In phase one, we compare a number of learning techniques (minimum square error linear regression, boosting, support vector machines), and determine which one is best suited to the problem by looking at its average cross validation error rate on test data. In phase two, we apply two dimensionality reduction techniques, namely subset selection and principal component analysis, to the best learner from phase one.

Our results show that the optimal technique based on accuracy (and ease of use) is the simple minimum square error procedure (multivariate linear regression via matrix pseudo-inverse), producing an average cross-validation test data error rate of 8.90% (accuracy of 91.1%) when using a subset of 9 of the 36 features.

2 Related Work

Yang et al [11] applied Bayesian learning techniques to build a model that predicts the winning percentage of a professional baseball team. The authors built a two stage Bayesian model based on the following predictor variables (parameters) : the past performance (winning percentage), the batting ability (batting averages), the strength of starting pitchers, and the frequency of home field advantage. The authors assumed that the effect of the other attributes is minor. The first three variables are combined into a single measure of team strength, and the first stage models the probability (α) of a team winning an individual game as a random draw from a beta distribution. The second stage is a random draw from a Bernoulli distribution given α. The authors then derive the predictive distribution for the probability of a team winning a game given past data:

$$p(X = x \mid Data) = C \int p(X = x \mid \alpha)\, p(\alpha \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta} \mid Data)\, d\boldsymbol{\theta}\, d\alpha$$

where $X \in \{0, 1\}$ is a random variable indicating a win (1) or loss (0), C is a normalizing constant, and $\boldsymbol{\theta}$ is the parameter vector. Gibbs sampling [13] (a form of Markov Chain Monte Carlo sampling) was used to a obtain posterior estimate of the distribution, which was used to predict team winning percentages starting from four different times during the 2001 season. Their results show that the model was more effective in predicting division winners (the teams with the highest winning percentages at the end of the season) the later the starting point in the season (i.e. with more past data). However, the contributions of each of the predictor variables were not so easy to pin down, and although the authors confirm "the difficulty in determining the exact contribution of the [past] winning percentage, batting, and pitching to the prediction problem", they hypothesize that the strength of starting pitchers is a key feature in prediction.

Barry et. al [12] attacked the same problem of predicting division winners given any starting time point during the season. They constructed a Bayesian model, derived a form for the predictive distribution, and also used MCMC sampling to obtain an estimate of the predictive distribution. Their model allows for team strengths to change over time, however, it does not incorporate the strengths of starting pitchers. The authors argued that the effect of starting pitchers quickly averages out since starting pitchers work in a rotation fashion, i.e. a team's best pitcher does not start every game. The results are similar to the results shown by Yang et al [11] in that prediction accuracy increases as the season progresses.

3 Methodology

3.1 Data

The sample data used in this study was obtained from a very complete database available from www.baseball1.com. This database contains pitching, hitting, and fielding statistics for Major League Baseball from 1871 through 2004. It includes data from the two current leagues (American and National), the four other "major" leagues (American Association, Union Association, Players League, and Federal League), and the National Association of 1871-1875. The data was provided as a series of batting, pitching, and fielding tables for each player. These tables were combined into a "teams" table, in which each row corresponds to one team in one particular year, and the values in the row are the sum totals of the players' data for that team in that year.

3.1.1 Collection and Preprocessing

We took a subset of this data as our data set, specifically data for 34 years (1971 - 2004) for approximately 30 teams in the American League and the National League (the number of teams varied from year to year). This provided us a sample data set of 910 data points (448 positive instances and 462 negative instances), each data point being a vector of 36 features that summarized a team's performance in a given year : $\mathbf{x}_{team,year} = [feature_1, feature_2, ..., feature_{36}]$.

We performed two normalizations of the sample data set. First, since the total number of games played in a year has varied, we normalized all feature values by the number of games a team played in that year. Thus each raw feature value became an average-per-game value. The second normalization was a bit more subtle, and it was an attempt to eliminate the dependency of an individual team's performance on the performance of other teams in a given year. Since we are looking at features such as number of hits, number of strike-outs, etc., a situation could arise where in one year all teams were very good in pitching, and thus the number of hits by *all teams* (including winners) that season was low, whereas in another year all teams were poor in pitching, and thus the number of hits by *all teams* (including losers) was high. If we were just looking at the number of hits per game, this situation would mislead us into classifying some losers as winners, and some winners as losers. Therefore, we normalize each average-per-game feature value by the sum total of all teams' values for that feature in that given year, as shown in equation 1 below. This gives us a relative measure of a team's performance in a given year versus the rest of the teams in that year.

$$\mathbf{x}_{team,year,feature} = \frac{\mathbf{x}_{team,year,feature}}{\sum_{t}^{all\ teams} \mathbf{x}_{t,year,feature}} \quad , \ feature \in \{1, 2, ..., 36\} \qquad (1)$$

3.1.2 Training and Test Sets

The sample data set of 910 samples was divided into three non-overlapping data sets for the purposes of 3-fold cross validation. The data sets were labeled A, B, and C, where A = data for 1971 - 1982 (300 samples), B = data for 1983 - 1994 (316 samples), and C = 1995 - 2004 (294 samples). Thus when we refer to data set AB for example, we refer to the combined set of A and B, with 616 samples total.

3.2 Learning Techniques

Below we describe three learning techniques used to attack this problem. We initially applied the learning algorithms to the data utilizing all 36 features, in order to determine which technique yields the best results. Then, we experimented with feature reduction techniques to determine a smaller subset of features, with approximately equivalent (or better) predictive power compared to the full feature set.

Each learning technique was run using 3-fold cross validation, and measurements of training and test error (percent mis-classified) were collected for each fold, and the average cross-validation training and test error rates were computed for each learning technique.

Training Data Set	**Training Set Size**	**Test Data Set**	**Test Set Size**	**Training Error**	**Test Error**
AB	616	C	294	% of AB mis-classified	% of C mis-classified
AC	594	B	316	% of AC mis-classified	% of B mis-classified
BC	610	A	300	% of BC mis-classified	% of A mis-classified
				Average CV Error	**Average CV Error**

Table 1: Training and test data sets, and definitions of training and test errors.

3.2.1 Minimum Square Error (MSE) Classification

This is the simplest technique of the three, and it was chosen primarily for its simplicity. This technique is outlined in [2], and we briefly summarize it below. This technique trains a linear classifier via a matrix pseudo-inverse calculation, and is basically equivalent to a multivariate linear regression model. We learn the equation for a hyperplane that minimizes the sum of squared errors between the true value $\mathbf{y}$ and the predicted value $\mathbf{Xw}$. Specifically, we seek a $\mathbf{w}$ vector that minimizes the loss function $\mathbf{L}$:

$$\mathbf{L} = \|\mathbf{X}_{train}\mathbf{w} - \mathbf{y}_{train}\|_2^2 \tag{2}$$

where $\mathbf{X}_{train}$ is a n-by-d (rectangular) matrix of n (training) data samples of d features, and $\mathbf{y}_{train}$ is a vector of +1 and -1 truth values (+1 = winning team, -1 = losing team). The $\mathbf{w}$ that minimizes 2 can be found by taking the gradient of $\mathbf{L}$ w.r.t. $\mathbf{w}$ and setting it equal to zero : $\boldsymbol{\nabla}\mathbf{L}(\mathbf{w}) = \mathbf{X}_{train}^T(\mathbf{X}_{train}\mathbf{w} - \mathbf{y}_{train}) = \mathbf{0}$, yielding :

$$\mathbf{w} = (\mathbf{X}_{train}^T\mathbf{X}_{train})^{-\mathbf{1}}\mathbf{X}_{train}^T\mathbf{y}_{train} \tag{3}$$

The $\mathbf{w}$ vector is then used by to predict on a test data point $\mathbf{x}$ by computing $\hat{y}=sign(\mathbf{w}.\mathbf{x})$, $\hat{y} \in \{-1, 1\}$, or in matrix form :

$$\hat{\mathbf{y}} = sign(\mathbf{X}_{test}\mathbf{w}) \tag{4}$$

We implemented code to perform these calculations in the Maple 9 software package. The methodology was quite simple : for each data fold, compute $\mathbf{w}$ using equation (3), use equation (4) to predict, and measure the training and test error. The results were surprisingly good - an average cross-validation error rate of 10.33% was achieved on held out test data; see Experimental Results section below for details on each of the MSE runs.

3.2.2 Boosted Decision Tree Stumps

The results from the MSE strategy were quite good, but we wanted to see if we could improve by using more sophisticated methods, and the first sophisticated method we chose was boosting (specifically AdaBoost) using decision tree stumps as weak learners. A decision tree stump is a decision tree with only

two leaf nodes, and classification is done based on the value of a single attribute. We were inspired to try boosting due to its ability to achieve a 0% error rate on training data (given that good weak learners can be consistently generated), and by its resistance to overfitting. We were also encouraged by a comment by Breiman (NIPS Workshop, 1996), where he referred to AdaBoost with trees as the "best off-the-shelf classifier in the world" [1].

We implemented a version of the AdaBoost.M1 algorithm found in [1] using Java. Information on how to obtain the source code is in Appendix B. The AdaBoost.M1 algorithm we implemented is shown below in pseudocode. We ran the algorithm for M = {2,3,...29} to determine the optimal number of boosting iterations based on the error rate on held-out test data. The optimal number of boosting iterations in each fold was ultimately determined by the error rate of the generated weak learners (decision stumps) in each iteration. Once the error rate of a weak learner hit 50%, no further improvement in the test error rate could be expected since $\alpha_m = \log((1 - err_m)/err_m) = \log((1 - .5)/.5) = 0$, eliminating that weak learner from the master classifier, and stopping any further updates to the weights w_i of the instances.

AdaBoost.M1.

1. initialize all sample weights $w_i = 1/N,\ i = 1, 2, ...N$.
2. for m = 1 to M (M = number of boosting iterations)
   (a) Get the best weak learner G_m based on the training data and its weights w_i
   (b) Compute $err_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$
   (c) Compute $\alpha_m = \log((1 - err_m)/err_m)$
   (d) Update weight $w_i = w_i \exp[\alpha_m I(y_i \neq G_m(x_i))],\ i = 1, 2, ...N$.
3. Measure performance on training and test data after M iterations using
   the master classifier $G(x) = sign[\sum_{m=1}^{M} \alpha_m G_m(x)]$

The weak learners G_m were decision tree stumps which classified (split) the data based a threshold value of one of the 36 features. The features were real values. At each boosting iteration, a new decision stump is generated (line 2(a) above) by calculating the optimal feature as the basis for the split and its optimal threshold split value. The optimal split feature and its value are the ones that minimize the entropy (cost) of the data due to the split. An entropy calculation is performed over all possible splits, i.e. over all possible values (in the training data set) of all features. The entropy of a split (equation (5) is defined as [4] :

$$Entropy[split] = \sum_{i\ outcomes} p_i \lg(\frac{1}{p_i}) \tag{5}$$

In our case *outcomes* $\in$ *{rhsplus,rhsminus, lhsplus,lhsminus}*, specifying how many plus (+1) and minus (-1) classifications landed on the right hand side leaf and left hand side leaf, and p_i is the proportion of such outcomes at each leaf.

The pseudo-code for determining the best split feature and value is shown below. Note that since this version of AdaBoost.M1 modifies the weights of each incorrectly classified sample (line 2(d) above), the calculation of the optimal split feature and split value must take these weights into account. This is done in lines 13-14 and 17-18 below, by summing up the weights of the plus and minus instances at each leaf of the decision tree.

Optimal Split Calculation Pseudo-code.

```
1    // for each feature column...
```

```
2     for (int j = 0; j < data.numFeatures; j++)
3     {
4         // for each possible feature value from each of the N samples
5         for (int i = 0; i < data.Nsamples; i++)
6         {
7               // candidate split
8               double split = data.X[i][j];
9               // determine how this candidate split classifies the samples
10              for (int k = 0; k < data.Nsamples; k++)
11              {
12                  if (data.X[k][j] > split) {
13                      if (data.Y[k] == 1) rhsplus = rhsplus + weights[k];
14                      else rhsminus = rhsminus + weights[k];
15                  }
16                  else {
17                      if (data.Y[k] == 1) lhsplus = lhsplus + weights[k];
18                      else lhsminus = lhsminus + weights[k];
19                  }
20              }
21              // calculate entropy (cost) of this split, and then determine if its the lowest seen so far
22              cost = calculateCost (rhsplus, rhsminus, lhsplus, lhsminus)
23              if (cost < mincost[j]) {
24                  mincost[j] = cost;
25                  bestSplitValue[j] = split;
26              }
27          }
28          // if split on feature j is better that we've seen so far, store it
29          if (mincost[j] < bestFeatureSplitCost) {
30              bestFeatureSplitCost = mincost[j];
31              bestSplitFeatureIndex = j;
32              bestSplitFeatureValue = bestSplitValue[j];
33          }
34    }
```

A final implementation note : all of the decision stump tests initially started out as tests of the form "if x > threshold then classify +1 else classify -1". However, if an optimal split of this form yielded a decision tree with error > .5, then for this particular weak learner G_m, we reversed the direction of the test to "if x $\leq$ threshold then classify +1 else classify -1".

Alas, boosted decision stumps did not fare as well as the MSE approach, achieving an average cross-validation error rate of 16.72% on held out test data, and never reaching 0% error on training data. See the Experimental Results section below for specific details and analysis on each of the boosting runs.

3.2.3 Support Vector Machines (SVM)

Having been let down by boosted stumps, we decided to attempt classification using a support vector machine (SVM). A support vector machine (SVM) maps instance vectors from an "instance space" of

(lower) dimension d to "feature space" of (higher) dimension f (f $>$ d), via the use of a kernel function K [3]. The key hope behind this mapping is that if the instances are not linearly separable in the lower dimensional instance space (i.e. no linear hyperplane can be found that perfectly separates the plus instances from the minus instances), then this mapping will allow a separating linear hyperplane in the higher dimensional space.

Based on the training error rates of the MSE and boosted stumps techniques above, and additional training error rates from a perceptron training algorithm (see Appendix B), we hypothesized that the training data were not linearly separable in the instance space of 36-dimensions. The hope was that an SVM approach based on a non-linear mapping to a higher dimensional feature space would produce a more effective classifier than the simple MSE technique, and that it could achieve a training error rate near 0% and near perfect test error rates.

SVMs solve an optimization problem to determine the equation of a hyperplane ($\mathbf{w}$ vector) that best separates the data. If the data are not linearly separable, SVMs solve the so-called "soft-margin" optimization problem shown in equation (6), where C is the penalty for classification errors [6]. If the data are linearly separable, $C = 0$ and $\xi_i = 0$ in equation (6), and the problem becomes the so-called "hard-margin" optimization problem.

SVM optimization problem : solve

$$\min_{w,b,\xi} \left\{ \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{N}\xi_i \right\} \tag{6}$$

subject to the following constraints :

$$\begin{array}{ll} y_i(\mathbf{w}^T\phi(\mathbf{x}_i) + b) \geq 1 - \xi_i, & \forall i \\ \xi_i \geq 0, & \forall i \end{array}$$

The function $\phi(\mathbf{x})$ maps an instance $\mathbf{x}$ to a higher dimensional feature space, and $\phi(\mathbf{x})$ is ultimately defined by a kernel function $K(\mathbf{u}, \mathbf{v})$ such that the following property holds : $\phi(\mathbf{u})\phi(\mathbf{v}) = K(\mathbf{u}, \mathbf{v})$. We chose a non-linear mapping in the form of a radial basis function (RBF) kernel shown in equation (7), performing a number of runs to optimize the C and γ parameters (the two parameters one can adjust when using a RBF kernel).

$$K(u, v) = e^{-\gamma\|u-v\|^2} \tag{7}$$

The SVM tools used were the open-source packages SVMlight 6.01 [7] and LibSVM 2.71 [5] [6]. The reason two packages were used was that LibSVM provides an easy to use, out-of-the-box, parameter optimization script (in Python 2.3) for determining the best C and γ parameters for an RBF kernel (equation (7)). LibSVM also provides easy to use scripts to scale the input features to the range $[-1, 1]$. Scaling features is important in SVMs since it prevents features with large numeric values from overwhelming features with small numeric values. SVMlight was used since it can produce verbose summaries of runs.

Our methodology was as follows :

1. Use LibSVM to scale the training and test data sets.

2. Perform cross-validation training and test runs using SVMlight, using the default parameters for an RBF kernel. Measure training and test error.

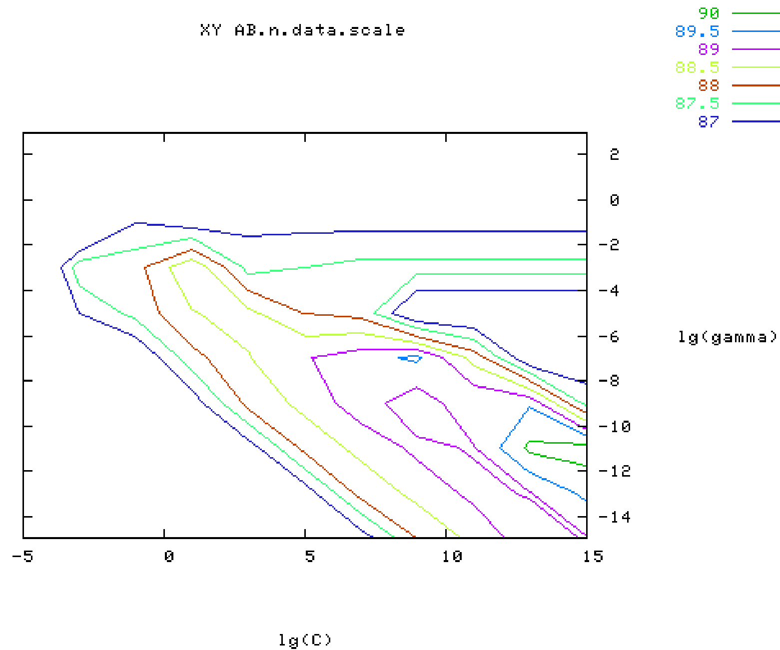

Figure 1: Grid search for optimal values of parameters C and γ using LibSVM. The graph shows level curves corresponding to constant levels of cross-validation error on training data.

3. Use LibSVM to determine the best C and γ parameters based on each training data set. LibSVM performs a grid-search on the parameter space of C and γ, searching for the values of C and γ that minimize the 10-fold cross-validation error on the training data set (see figure 1). The motivation behind this parameter search is to avoid overfitting : we approximate the yet unseen test data with a 10-fold cross validation of the training data and choose parameter values that perform best on this approximation.

4. Perform cross-validation training and test runs using SVMlight, this time using the optimized C and γ values. Measure training and test error.

However in spite of these attempts, the average cross-validation test data error rate did not change appreciably from that achieved using the simple MSE technique. See the Experimental Results section below for specific details and analysis of the SVM training runs.

3.3 Feature Selection Techniques

Having determined that the simple MSE technique very nearly produces the best results and is also the simplest to implement, we wanted to know if similar results could be obtained with a smaller set of features or a smaller set of linear combinations of features. Feature selection is useful because it helps remove irrelevant (and possibly noisy) features from the data set, thus improving accuracy, and it helps reduce

overall computation time. The two techniques used to perform feature selection were *subset selection* and *principal component analysis.*

3.3.1 Subset Selection

Subset selection is an iterative method in which a subset of features is chosen, the classifier is trained using this subset, and the performance (error rate) of the classifier on test data is measured. If the error rate on test data is sufficiently close (or better) to the test error rate when using the larger feature set, this subset is chosen as the current best set of features. The procedure is repeated until the accuracy on test data has dropped below some pre-defined threshold.

An exhaustive search of the subset space is very costly. The number of learning and testing experiments that need to be performed is on the order of 2^d, where d is the number of features. A simpler approach requiring at most d experiments is called *backward stepwise selection* [1]. Backward stepwise selection starts with the full set of features, and iteratively removes features, one by one. To determine which feature to remove at any one point, one can use the F-statistic (under certain assumptions - see below), which measures the change in the residual sum-of-squares (RSS) per feature removed :

$$RSS(\mathbf{w}) = \sum_{i=1}^{N}(\hat{y}_i - y_i)^2 = \sum_{i=1}^{N}(\mathbf{x}_i\mathbf{w} - y_i)^2. \tag{8}$$

The F-statistic is shown in equation (9) :

$$F = \frac{RSS(\mathbf{w}_{new}) - RSS(\mathbf{w}_{old})}{RSS(\mathbf{w}_{new})/(N-k-2)} \tag{9}$$

where $\mathbf{w}_{new}$ refers to the new solution vector (equation (3)) obtained after eliminating one feature from the training data set $\mathbf{X}$, $\mathbf{w}_{old}$ refers to the previous solution vector obtained from the larger feature set, and k is the current number of features in the model.

In backward stepwise selection, we eliminate the feature producing the smallest F value at each stage, and we stop when each feature, if eliminated, produces an F value which is greater than the 95th percentile of the $F_{1,N-k-2}$ distribution. In our case, this stopping value is 3.84. Therefore our algorithm for subset selection is as follows :

Backward Stepwise Selection :
for k = 1 to (number of features in full set - 1) and (not stopRun) do
    $w_{old} = (\mathbf{X}_{train}^T\mathbf{X}_{train})^{-1}\mathbf{X}_{train}^T\mathbf{y}_{train}$
    for i = 1 to columns[$\mathbf{X}_{train}$] do
        $\mathbf{M}$ = SubMatrix[$\mathbf{X}_{train}$] with the ith column eliminated
        $w_{new} = (\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}\,\mathbf{y}_{train}$
        $F = \frac{RSS(\mathbf{w}_{new}) - RSS(\mathbf{w}_{old})}{RSS(\mathbf{w}_{new})/(N-k-2)}$
        if (F < lowest_F_seen) then
            lowest_F_seen = F
            i_lowest = i
            $\mathbf{w}_{lowest} = \mathbf{w}_{new}$
        end if

```
        end do
        if (lowest_F_seen < 3.84)
            X_train= SubMatrix[X_train] with the i_lowest column eliminated
            train_Predictions = sign(X_train w_lowest)
            Calculate number of training errors
            X_test= SubMatrix[X_test] with the i_lowest column eliminated
            test_Predictions = sign(X_test w_lowest)
            Calculate number of test errors
        else
            stopRun = true
        end if
    end do
```

For a detailed explanation of why the F-statistic and F-distribution are appropriate, please see [1]. However, intuitively we can motivate it as follows. The F-distribution is defined as a ratio of two chi-square distributions [8]. Furthermore, if a random variable y has a normal distribution with mean μ and variance σ^2, then the distribution of $\frac{n}{\sigma^2}S^2$ (where the sample variance $S^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2$ is the biased estimator of σ^2) has a chi-square distribution with $(n-1)$ degrees of freedom : $\frac{n}{\sigma^2}S^2 \sim \chi^2_{n-1}$. Therefore we have that $\frac{1}{\sigma^2}\sum_{i=1}^{n}(y_i - \bar{y})^2 \sim \chi^2_{n-1}$. If we make some assumptions about the distribution of the true value of y that we are trying to predict (e.g. it has a fixed variance σ^2), we see that a ratio of two chi-square distributions, which is an F-distribution, is actually *a ratio of the sample variances* of those two distributions. Furthermore, those sample variances are proportional to the residual sum-of-squares (RSS, equation (8)). So the change of the RSS values from one model to the next computed by equation (9) is proportional to the change in sample variance of y from one model to the next. When this change in variance is small we can say that the new model (e.g. the one with the smaller feature set) is a good approximation to the old model, and thus we can eliminate the held out feature(s) from the model. When this change in variance becomes too great regardless of which feature we eliminate, we know only key features are left in the model, and thus we stop.

3.3.2 Principal Component Analysis

The second dimensionality reduction technique we attempted was principal component analysis (PCA). PCA seeks to explain the variance-covariance relationships of a large set of variables through a relatively smaller number of linear combinations of these variables [9]. The principal components of a data set are those linear combinations of the original features that account for most of the variability (information) in the data, and thus the high dimensional data can be approximated by a lower dimensional set of principal components. In effect we are projecting each data vector $\mathbf{x}$ in $\mathbb{R}^d$ onto a subspace $\mathbb{R}^k$ spanned by the first k principal components, where $(k < d)$. It can be shown that the first k principal components are the k eigenvectors corresponding to the first k largest eigenvalues of the covariance matrix of the data $\mathbf{X}$ [2] [9].

Therefore, our algorithm for analyzing principal components was as follows [10] :

Principal Components Analysis Algorithm.

1. Compute the mean vector and covariance matrix of the training and test data sets.

$$\mathbf{m}_{train} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i^{train} \qquad cov(\mathbf{X}_{train}) = \frac{1}{N-1}\sum_{i=1}^{N}(\mathbf{x}_i^{train} - \mathbf{m}_{train})^T(\mathbf{x}_i^{train} - \mathbf{m}_{train})$$

$$\mathbf{m}_{test} = \frac{1}{M}\sum_{i=1}^{M}\mathbf{x}_i^{test} \qquad cov(\mathbf{X}_{test}) = \frac{1}{M-1}\sum_{i=1}^{M}(\mathbf{x}_i^{test} - \mathbf{m}_{test})^T(\mathbf{x}_i^{test} - \mathbf{m}_{test})$$

2. For k = 1 to number_of_features do :
   a. Compute the k eigenvectors $\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_k$ corresponding to the k largest eigenvalues of $cov(\mathbf{X}_{train})$, and form the matrix $\mathbf{E} = [\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_k]$.
   b. Subtract the mean vector $\mathbf{m}_{train}$ from each data point $\mathbf{x}$ in $\mathbf{X}_{train}$, yielding $\mathbf{X}_{train-m}$, and do the same for $\mathbf{X}_{test}$.
   c. Compute the projection of each training and test data point $\mathbf{x}$ onto the space spanned by the k eigenvectors by :
      $\mathbf{X}_{train_p} = \mathbf{X}_{train-m}\,\mathbf{E}^T$
      $\mathbf{X}_{test_p} = \mathbf{X}_{test-m}\,\mathbf{E}^T$
   d. Train a classifier (using MSE) on this reduced dimensionality $\mathbf{X}_{train_p}$, i.e. compute $\mathbf{w} = (\mathbf{X}_{train_p}^T \mathbf{X}_{train_p})^{-1}\mathbf{X}_{train_p}^T\,\mathbf{y}_{train}$.
   e. Predict using test data using $\hat{\mathbf{y}} = \mathbf{X}_{test_p}\,\mathbf{w}$, and calculate the test error.
3. Repeat steps 1 and 2 for all folds in 3-fold cross validation.

Having done the above for k = {1,2,....d}, where d is the total number of features, we plot the average cross validation test and training errors versus the number of principal components k (see Experimental Results section for plots and analysis).

4 Experimental Results

4.1 Baseline

Given that we had 448 positive instances and 462 negative instances in the full data set, a null learning rule could be : predict negative all the time. The error of this rule would be 49.2%, and so this is the baseline against which we compare our results.

4.2 Full Feature Set

Tables 2 and 3 below compare the results of the three learning techniques using the entire set of features. As one can see from Table 3, the average cross validation test error rate of the simple MSE technique was quite good (10.33%), and was nearly identical to the average test error rate of the more complicated SVM technique using optimized parameters (10.19%).

The interesting thing to note about the SVM results when comparing training error rate vs. test error rate is that the SVM with the default RBF parameters definitely overfit the data : average training error was 0.49% whereas average test error was 21.89%. However, using optimized RBF parameters, we sacrificed a bit on the training error rate (4.81%), but achieved a large improvement in the test error rate (down to 10.19%). Therefore in this case, the LibSVM parameter optimization techniques (described above) do help avoid overfitting.

Training Data Set	**MSE**	**Boosted Stumps**	**SVM - Default RBF Parameters**	**SVM - Optimized RBF Parameters**
AB	6.49 %	9.25 %	0.49 %	3.57 %
AC	6.90 %	10.44 %	0.51 %	7.58 %
BC	5.41 %	6.89 %	0.49 %	3.28 %
Average	**6.27 %**	**8.86 %**	**0.49 %**	**4.81 %**

Table 2: Comparison of Training Error Rate (the complete feature set).

Test Data Set	**MSE**	**Boosted Stumps**	**SVM - Default RBF Parameters**	**SVM - Optimized RBF Parameters**
C	9.18 %	19.39 %	15.31 %	8.50 %
B	9.81 %	16.67 %	18.04 %	11.39 %
A	12.00 %	14.33 %	32.33 %	10.67 %
Average	**10.33 %**	**16.83 %**	**21.89 %**	**10.19 %**

Table 3: Comparison of Test Error Rate (the complete feature set).

The boosting results also show symptoms of overfitting. The average training error is around 9%, but the average test error is nearly twice that, at around 17%. The optimal number of boosting iterations run for each fold was determined by monitoring the error rate on held out test data vs. the number of boosting iterations (see figure 3), and these are the error rates depicted in Tables 2 and 3 above. The training error rate for boosting runs is shown in figure 2 - note how the training rate plateaus in the range of 6% - 10%, and it never reaches 0%, an indication that after a while, the weak learners could do no better than random guessing (50% error rate).

4.3 Reduced Feature Set

4.3.1 Subset Selection

The algorithm and the methodology of backward stepwise subset selection is depicted graphically in figure 4 for one of the cross-validation runs. In this run, feature removal was halted when the F-score exceeded 3.84, which left a subset of 9 features.

The complete results for cross-validation runs are shown in Table 4 below. Note that compared to the MSE values from Table 2 and Table 3, our training error has increased a bit from 6.27% to 8.30%, but our test error has decreased from 10.33% to 8.90%, indicating that we may have had a some overfitting with all 36 features, and we that we were able to remove some of that effect with 7 or 9 features. The subset of features chosen in each cross-validation run was not identical, although there were many common features in each of the subsets.

4.3.2 Principal Component Analysis

Figure 5 shows the training and test error rates as a function of the number of principal components used by the MSE algorithm. Although PCA is successful in reducing the dimensionality of the instance

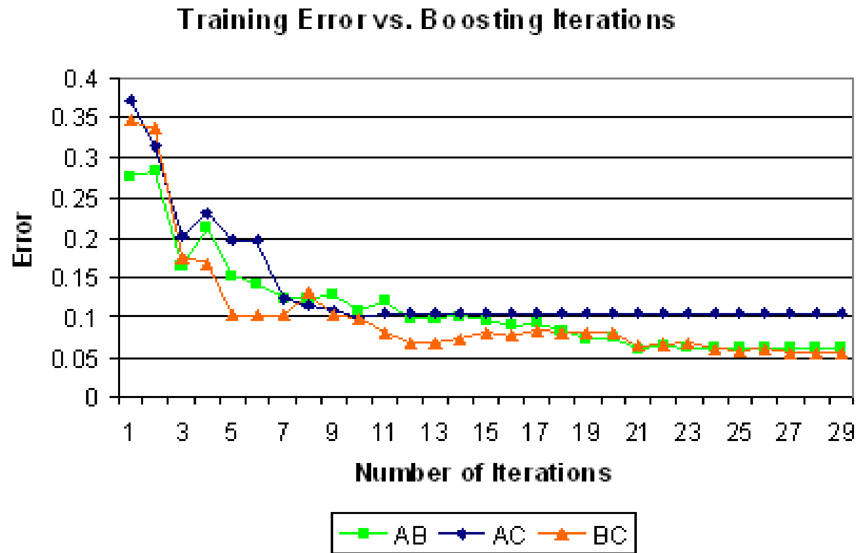

Figure 2: Training error rate (% misclassified) as a function of boosting iterations on the three training data sets : AB, AC, BC. Note that the training error rate does not reach 0% due to the fact that the weak learners' error rates hit 50%.

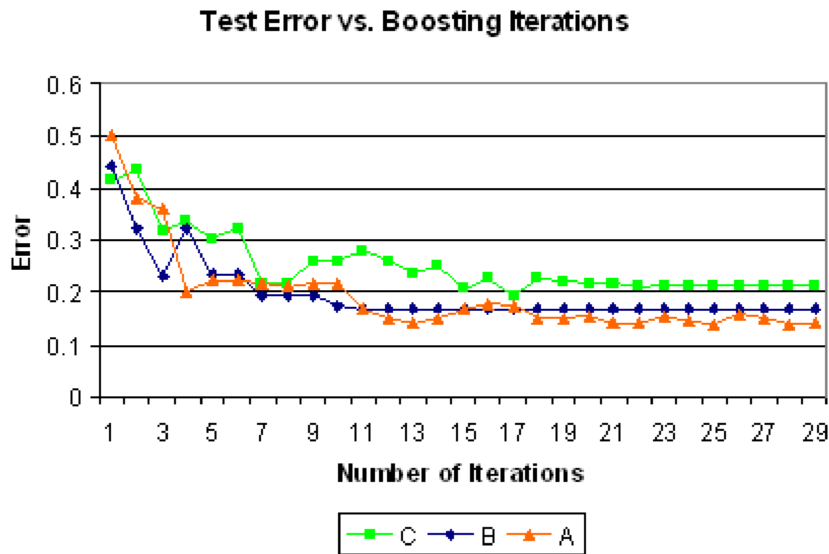

Figure 3: Test error rate (% misclassified) as a function of boosting iterations on the three training data sets : C, B, A. The optimal number of boosing iterations for each set was : 17, 11, 13, respectively.

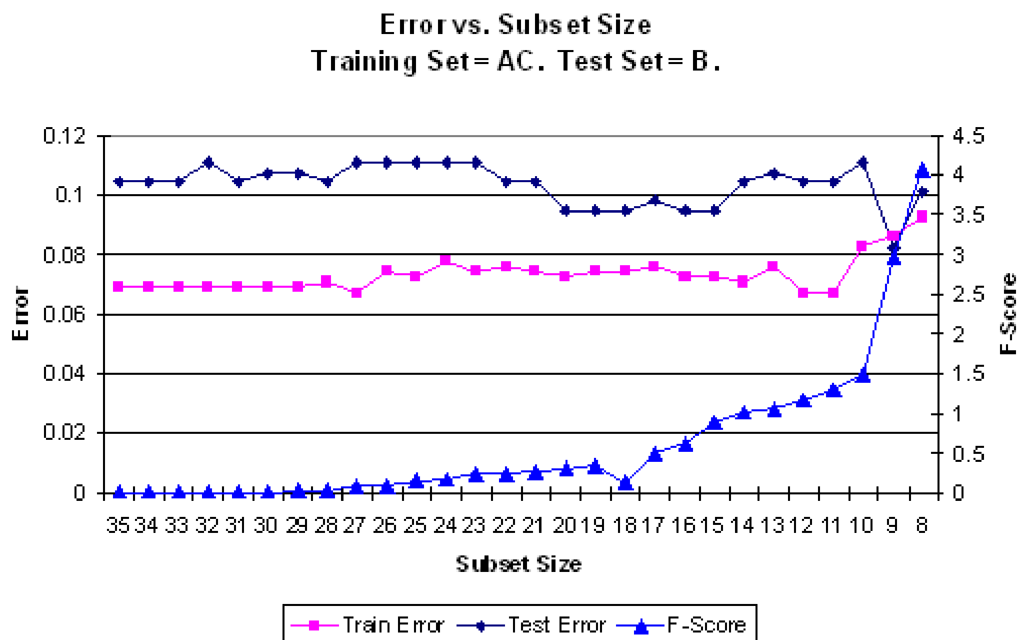

Figure 4: Left y-axis shows error rates as a function of the number of features in the subset. Right y-axis shows the F-statistic (F-score) value for each subset. Feature removal was halted when the F-score exceeded 3.84.

Training Data Set	**Test Data Set**	**Subset Size**	**Training Error**	**Test Error**
AB	C	9	9.42 %	6.46 %
AC	B	9	8.59 %	8.23 %
BC	A	7	6.89 %	12.00 %
Average			**8.30 %**	**8.90 %**

Table 4: Training and test errors using the MSE technique with a smaller subset (size 9) of features.

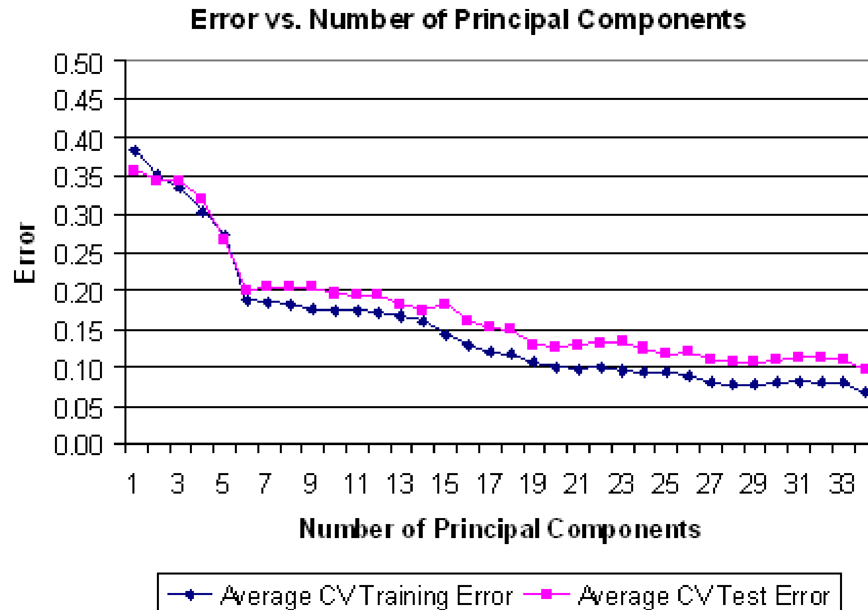

Figure 5: Training and test error rates (using the MSE technique) as a function of the number of pricincipal components. Note how the error rate flattens out after the 6th principal component.

Model A	Model B	Model C
8.57 %	8.46 %	8.46 %

Table 5: Error rate of models on entire data set.

space, it suffers two drawbacks in this case : one, it does not perform as well as the backward stepwise selection method (test error rate is about 20% using 9 principal components, compared to about 8% for subset selection), and two, since each principal component is a linear combination of the 36 features, PCA results are very difficult to interpret if one is looking for the essential individual features that separate winners from losers.

5 Conclusions and Open Problems

5.1 Final Model

So now the question is : which model shall we choose and stick with as a final predictor of whether a team's winning percentage is above 50% in a given year ? Based on the feature selection results above, it is clear that subset selection performs better and is easier to interpret than PCA. Given that, we can compare how the different models obtained from the 3-fold cross validation runs perform on the entire data set, and choose the model with the lowest error rate [1]. Table 5 below shows the performance of each model on the entire data set of 910 samples (model C = the model obtained via training on data set AB, etc.).

It is a tie between Model B and Model C. Model B is the linear combination of (*normalized*) features

shown in equation (10). See Appendix A for a full description of features. The features prefixed by "b_" are batting related features (offense), the features prefixed by "p_" are pitching (defense) related, and the features prefixed by "f_" are fielding related features.

$$\begin{aligned} Model\ B(x) &= -14.002(\overline{b_AB}) + 0.0661(\overline{b_R}) + 3.405(\overline{b_H}) + 0.134(\overline{b_HR}) - 0.129(\overline{b_CS}) \\ &\quad -0.085(\overline{b_SH}) - 0.232(\overline{b_GIDP}) + 11.020(\overline{p_IPOuts}) - 0.771(\overline{p_R}) \end{aligned} \tag{10}$$

The rule as always is : predict winner if $B(x) > 0$. Model B says that a team's winning percentage is **negatively** correlated with the following normalized quantities : the number of "at bats" (b_AB) by the team, the number of times the team's players were caught stealing (b_CS), the number of sacrifice hits (b_SH), the number of times batters grounded into double plays (b_GIDP), and the number of runs allowed by its pitchers (p_R). It is **positively** correlated with : the number of runs scored (b_R), the number of hits (b_H), the number of home runs (b_HR), and the number of "outs" pitched by pitchers (p_IPOuts, equals 3 x innings pitched). Most of these correlations were to expected *except* the negative correlation to the number of "at bats" and the *positive* correlation to the number of outs pitched. One would expect that the more batting chances a team had, the more games it would win, and the more pitches (outs) a team had to throw, the more games it would lose.

Model C is shown in equation (11). It exhibits similar coefficients, and shares 7 of 9 features with Model B. It uses number of runs batted in (b_RBI) instead of number of home runs (b_HR), and number of putouts (e.g. defensive outs, f_PO) instead number of outs pitched (p_IPOuts). It still exhibits the strange negative correlation on the number of at bats. We believe that a positive correlation with the number of putouts makes more sense versus number of outs pitched, and combined with the fact that Model C had the best performance on unseen test data (see Table 4), **we choose Model C as our final model**.

$$\begin{aligned} Model\ C(x) &= -15.268(\overline{b_AB}) + 3.486(\overline{b_H}) + 1.085(\overline{b_RBI}) - 0.136(\overline{b_CS}) - 0.122(\overline{b_SH}) \\ &\quad -0.179(\overline{b_SF}) - 0.274(\overline{b_GIDP}) + 0.705(\overline{p_R}) + 12.111(\overline{f_PO}) \end{aligned} \tag{11}$$

5.2 Boosting Model Comparison

Although boosting did not perform as well as the other techniques, it is interesting to compare one of the models generated from the boosting runs with the linear models above. The boosting version of Model B (after 11 iterations) is shown below.

$$\begin{gathered} G(x) = 0.523(p_R < 3.124) + 0.863(b_R >= 3.796) + 1.1245(p_R < 3.876) + 0.870(b_RBI >= 3.228) \\ +0.775(p_R < 3.137) + 0.653(b_RBI >= 4.055) + 0.885(p_SV >= 2.992) + 0.726(p_BB < 3.703) \\ +0.649(b_RBI >= 3.427) + 0.592(p_R < 3.071) + 0.388(p_HBP < 3.148) \end{gathered}$$

Each of the boolean clauses outputs +1 (true) or -1 (false). Again, we predict winner if $G(x) > 0$. Boosting seems to have focused on a smaller subset of features, all of which make intuitive sense : runs allowed (p_R), runs scored (b_R), runs batted in (b_RBI), saved games (p_SV), walks allowed (p_BB), and the strangest feature of them all : number of batters hit by pitches (b_HBP) !

5.3 Open Problems

A curious open problem arising out of this study is the performance of the SVM technique. What is it about this data that made the MSE technique comparable to an SVM technique, and why wasn't the

SVM markedly better than MSE ? In general, are there specific characteristics of data sets that make support vector machines not the optimal classifiers ? Weston et al [14] propose that SVMs perform poorly when many of the features are irrelevant. However, experiments using an SVM with the subset of nine features (obtained from feature selection) did not significantly change the performance of the SVM classifier (9.65% vs. 10.19% average error on test data). We believe this is a worthy area of future research.

A second open problem is the issue of predicting not only winning percentage, but being able to predict champions, e.g. league winners, of which there are only two per year. This is a much more difficult problem, but a much more lucrative one.

Finally, given the highly non-linear aspects of the data, we believe a neural network approach may also be a good direction to explore.

6 Appendix A - Description of Features

Feature Name	Definition
b_AB	At bats
b_R	Runs scored by batters
b_H	Hits by batters
b_2B	Doubles hit by batters
b_3B	Triples hit by batters
b_HR	Home runs hit by batters
b_RBI	Runs batted in
b_SB	Stolen bases by batters
b_CS	Batters caught stealing
b_BB	Base on balls
b_SO	Strikeouts by batters
b_IBB	Intentional walks of batters
b_HBP	Batters hit by pitch
b_SH	Sacrifice hits by batters
b_SF	Sacrifice flies by batters
b_GIDP	Batters grounded into double plays
p_CG	Complete games by starting pitchers
p_SHO	Shutouts pitched
p_SV	Saves (games) by pitchers
p_IPOuts	Outs pitched (innings pitched x 3)
p_H	Hits allowed by pitchers
p_ER	Earned runs allowed by pitchers
p_HR	Home runs allowed by pitchers
p_BB	Walks allowed by pitchers
p_SO	Strikeouts by pitchers
p_IBB	Intentional walks by pitchers
p_WP	Wild pitches
p_HBP	Batters hit by pitchers
p_BK	Balks
p_BFK	Batters faced by pitchers
p_R	Runs allowed by pitchers
p_ERA	Pitcher's earned run average
f_PO	Defensive put outs
f_A	Defensive assists
f_E	Defensive errors
f_DP	Defensive double plays

7 Appendix B - Other Experiments

Below are two quick and dirty experiments we ran on the data as well.

7.1 Perceptron

In order to get some idea if this data was at all linearly separable, we ran the Perceptron algorithm (stochastic gradient descent) as defined in [4]. Specifically, our weight update rule (component-wise) was :

for k = 1 to N do
    $w_i = w_i - (\eta/k)(x_{i,d})(\hat{y} - y)$

where $\eta = w_i = 1/N$ initially. Unfortunately, the results were quite poor : 27.72 % average cross validation training error, and thus we did not proceed further. This was mainly meant to be a quick check to see if the data were linearly separable, since if the data are linearly separable, the perceptron algorithm will converge. However, we did not see any evidence of convergence, leading us to strongly suspect (although we cannot confirm) that the data are not linearly separable.

7.2 Bayes Optimal

We fit the winners' and losers' training data to 36-dimensional Gaussian distributions (using Mathematica 5.1), attached prior probabilities to each class equal to the proportion of the class in the training data set, and used the Bayes discriminant [2] to classify the test data as shown in equation (12) :

$$g_i(\mathbf{x}) = \mathbf{x}^t \mathbf{W}_i \mathbf{x} + \mathbf{w}_i^t \mathbf{x} + w_{i0} \quad , i \in \{winners, losers\} \tag{12}$$

where
$\mathbf{W}_i = -\frac{1}{2}\boldsymbol{\Sigma}_i^{-1}$
$\mathbf{w}_i = \boldsymbol{\Sigma}_i^{-1}\boldsymbol{\mu}_i$
$w_{i0} = -\frac{1}{2}\boldsymbol{\mu}^t \mathbf{W}_i \boldsymbol{\mu} - \frac{1}{2}\ln|\boldsymbol{\Sigma}_i| + \ln P(\omega_i)$

The prediction rule on test data is : predict winner if $g_{winner}(\mathbf{x}) > g_{loser}(\mathbf{x})$. Performance on training data was excellent (the average CV training error was 5.00 %), and not surprisingly, it led to serious overfitting, producing poor test data results (21.17% average CV test error).

8 Appendix C - Source Code

Source code and the data files are available upon request by emailing vvonbrze@ucsc.edu. Samples of the source code for the two feature reduction techniques are included below.

8.1 Maple 9 Backward Stepwise Subset Selection Code

```
with(LinearAlgebra): train := "XY_BC.n.csv": test := "XY_A.n.csv":
  trainXY := ImportMatrix(train, source=delimited, format=rectangular, delimiter=","):
  XY := ImportMatrix(test, source=delimited, format=rectangular, delimiter=","):
  rowTrain := RowDimension(trainXY): colTrain := ColumnDimension(trainXY):
  rowX := RowDimension(XY): colX := ColumnDimension(XY):
  trainX := SubMatrix(trainXY,[1..rowTrain],[1..(colTrain-1)]):
  trainY := convert(SubMatrix(trainXY,[1..rowTrain],[colTrain]),Vector):
  X := SubMatrix(XY,[1..rowX],[1..colX-1]): Y := convert(SubMatrix(XY,[1..rowX],[colX]),Vector):
  stopRun := 0:
  for k from 1 by 1 while (k <= 34 and stopRun = 0) do
      # calc initial starting model residual sum of squares RSS
      invTrainX := Matrix(MatrixInverse(trainX,method=pseudo)):
      wpsTrain := Vector(invTrainX.trainY):
      RSS := Norm(trainX.wpsTrain - trainY,2):
      lowestF := 100: i_lowest := 0:
      for i from 1 by 1 while i <= (ColumnDimension(trainX)) do
          M := SubMatrix(trainX,[1..rowTrain],[1..(i-1),(i+1)..(ColumnDimension(trainX))]):
          invTrainX := Matrix(MatrixInverse(M,method=pseudo)):
          wpsTrain := Vector(invTrainX.trainY):
          F := (rowTrain - (colTrain-1 - k)-2)*(1-RSS/Norm(M.wpsTrain - trainY,2)):
          if (F < lowestF) then
              lowestF := F: i_lowest := i: wpsLowest := Vector(wpsTrain):
          end if:
      end do:
      if (lowestF < 3.84) then
          trainX := SubMatrix(trainX, [1..rowTrain], [1..(i_lowest-1), (i_lowest+1)..ColumnDimension(trainX)]):
          trainPred := trainX.wpsLowest:
          trainErrors := 0:
          for j from 1 by 1 while j <= rowTrain do
              if (sign(trainPred[j]) <> sign(trainY[j])) then
                  trainErrors := trainErrors + 1;
              end if;
          end do:
          X := SubMatrix(X, [1..rowX], [1..(i_lowest-1),(i_lowest+1)..ColumnDimension(X)]):
          pred := X.wpsLowest: numErrors := 0:
          for m from 1 by 1 while m <= rowX do
              if (sign(pred[m]) <> sign(Y[m])) then
                  numErrors := numErrors + 1;
```

```
            end if;
        end do:
    else  stopRun := 1:
    end if:   end do:
```

8.1.1 Mathematica 5.1 PCA Code

```
<<Statistics`MultiDescriptiveStatistics`;
   Xtrain = Import["X_AB.n.csv", "CSV"]; Ytrain = Import["Y_AB.n.csv", "CSV"];
   Xtest = Import["X_C.n.csv", "CSV"]; Ytest =Import["Y_C.n.csv","CSV"];
   trainingE = .; testE = .;
   trainingE = Table[0,{36}];
   testE = Table[0,{36}];
   meanXtrain = Mean[Xtrain];
   XtrainMod =Table[0,{Dimensions[Xtrain][[1]]},{Dimensions[Xtrain][[2]]}];
   For[i=1,i<=Dimensions[Xtrain][[1]], i++,{XtrainMod [[i]]=Xtrain[[i]] - meanXtrain;}];
   meanXtest = Mean[Xtest];
   XtestMod = Table[0,{Dimensions[Xtest][[1]]},{Dimensions[Xtest][[2]]}];
   For[i=1,i<=Dimensions[Xtest][[1]], i++,{XtestMod [[i]]=Xtest[[i]] - meanXtest;}];
   covXtrain = CovarianceMatrix[Xtrain];
   For[numevals=1,numevals<=36,numevals++,
   {
       evec =Eigenvectors[covXtrain,numevals];
       Xtrainproj =XtrainMod.Transpose[evec];
       w = PseudoInverse[Xtrainproj].Ytrain;
       trainpred = Xtrainproj.w;
       Xtestproj = XtestMod.Transpose[evec];
       testpred = Xtestproj.w;
       trainsum = 0;
       For[i=1,i<=Dimensions[Ytrain][[1]],i++,
       {
           If[trainpred[[i]].Ytrain[[i]]<0,trainsum++,]
       }
       ];
       testsum = 0;
       For[i=1,i<=Dimensions[Ytest][[1]],i++,
       {
           If[testpred[[i]].Ytest[[i]]<0,testsum++,]
       }
       ];
       trainingE[[numevals]]=trainsum/Dimensions[Ytrain][[1]] //N;
       testE[[numevals]]=testsum/Dimensions[Ytest][[1]] //N;
    }
    ]
```

References

[1] T. Hastie, R. Tibshirani, J. Friedman. *The Elements of Statistical Learning*, sections 3.4, 7.10, 10.1. Springer-Verlag, 2001.

[2] R. Duda, P. Hart, D. Stork. *Pattern Classification*, 2nd Edition, sections 2.6.2, 3.8.1, 5.8.1. Wiley-Interscience, 2000.

[3] C. Burgess. *A Tutorial on Support Vector Machines for Pattern Recognition.* Data Mining and Knowledge Discovery, Volume 2, Number 2, pages 121-167, 1998.

[4] T. Mitchell. *Machine Learning*, sections 3.4, 4.4. McGraw-Hill, 1997.

[5] LibSVM software package available from : http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[6] C. Hsu, C. Chang, C. Lin. *A Practical Guide to Support Vector Classification.* www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf.

[7] SVMlight software package available from : http://svmlight.joachims.org/

[8] R. Hogg, A. Craig. *Introduction to Mathematical Statistics*, 5th Edition, section 4.4. Prentice-Hall, 1995.

[9] R. Johnson, D. Wichern. *Applied Multivariate Statistical Analysis*, 5th Edition, section 8.2. Prentice-Hall 2002.

[10] D. Barber. *Learning from Data. Dimensionality Reduction: Principal Component Analysis.* www.inf.ed.ac.uk/teaching/ courses/lfd/lectures/lfd_2004_dim_red.pdf.

[11] T. Yang, T. Swartz. *A Two-Stage Bayesian Model for Predicting Winners in Major League Baseball*, Journal of Data Science, 2, 61-73.

[12] D. Barry, J. Hartigan. *Choice Models for Predicting Divisional Winners in Major League Baseball.* Journal of the American Statistical Association, 88, 776-774.

[13] A. Gelman, J. Carlin, H. Stern, D. Rubin. *Bayesian Data Analysis*, 2nd Edition. Chapman & Hall/CRC, 2004.

[14] J. Weston, S. Mukherjee, O. Chapelle, M. Pontil, T. Poggio, V. Vapnik. *Feature selection for SVMs.* NIPS 13, 2000.