Parsimonious regression

Nikhil Murali Kishore

September 13, 2015

Contents

Chapter 1

Abstract

Generally, the algorithms used for attribute selection in machine learning are either quite slow or neglect the interactions between the attributes. Solving the "Elastic Net" problem using the "Cyclic Coordinate Descent" algorithm can be a fast and effective way to perform attribute selection for linear models. The Elastic Net problem involves choosing coefficients to minimize the squared error while imposing a penalty on the coefficients. The penalty is a combination of the L1 and L2 norms of the coefficients. Solving the elastic net problem effectively performs attribute selection while building the model, so it does not neglect the interactions between the attributes. Moreover, Cyclic Coordinate Descent is a really fast procedure because it cycles over the coefficients, updating them one at a time, and converges quickly. Thus, it is easy to see that this approach is an excellent way to perform attribute selection. This project aims to build a WEKA classifier that solves the Elastic Net problem using Cyclic Coordinate Descent. Since it has already been implemented as an R package, the majority of the experiments in this report compare the R implementation with the WEKA implementation.

Chapter 2

Introduction

The goal of this project is to implement a machine learning algorithm in WEKA to solve the "elastic-net" problem for linear regression using coordinate descent.

The simplest algorithm for linear regression is called Ordinary Least Squares (OLS) regression. OLS fits a hyperplane through the data to minimize the total squared error, which is described as:

$F_{error} = \sum_{i=1}^{n}(y_i - \beta_0 - x_i^T\beta)^2$

Here, x_i is the i^{th} instance, β is the coefficient vector, β_0 is the intercept and y_i is the class. It is very fast and simple to minimize this function because the solution is directly derived from linear algebra and it relies on a few basic matrix operations.

OLS regression works well if there is a simple linear dependency between the class and the predictors, and when the number of training examples (n) is a lot higher than the number of predictors (p). Conversely, the main disadvantage of OLS is that it tends to overfit the data if there are too many predictors and too few examples.

The problem is very obvious in cases where the number of examples is less than the number of predictors. In this case, performing OLS regression is equivalent to solving a system of n linear equations with p variables where $p > n$. There can be infinitely many solutions in this case, so OLS is sure to overfit. OLS can also be wildly inaccurate if some of the predictors are noisy, since it necessarily assigns a coefficient for each predictor.

An effective solution to the problems with OLS is to try and find the simplest model that can explain the data to a reasonably high degree. The heuristic used here is that a simpler model is more likely to be correct even if it is marginally less accurate in predicting the class.

There are many ways to find a simple solution that explains the data. One of the ways is to add a penalty term to the total squared error. The objective function to be minimized now becomes (total squared error + penalty term). The penalty term is always designed to penalize higher values of coefficients, and the desired goal is to obtain a simple model by tuning out the spurious predictors (by assigning them very low coefficients).

Three types of penalties are commonly used: the L2 norm of the coefficients, the L1 norm or a combination of both. The "elastic net" penalty is the third type: it is a combination of the L1 and L2 norms.

The penalties depend on the coefficients β_j for $j \in \{1, 2, ...p\}$, and two parameters λ and α, such that $0 \leq \alpha \leq 1$ and $\lambda \geq 0$. The three penalty types are defined below:

- L1 penalty $= \lambda \sum_{j=1}^{p} \beta_j^2$
- L2 penalty $= \lambda \sum_{j=1}^{p} |\beta_j|$
- Elastic net penalty $= \lambda \sum_{j=1}^{p} (\frac{1}{2}(1-\alpha)\beta_j^2 + \alpha|\beta_j|)$

We see that the L1 and L2 penalties are just special cases of the elastic net penalty. The L2 penalty tends to shrink the coefficients of spurious predictors close to zero, but it never quite makes them 0. It also tends to shrink correlated predictors towards each other. The L1 penalty shrinks spurious predictors to zero. In case there are correlated predictors, it picks one and shrinks the rest to 0. The elastic net is a combination of the L1 and L2 penalties. For α close to 1, it behaves much like the lasso but does not ignore correlated predictors.

The disadvantage of introducing a penalty term is that it is no longer possible to minimize the objective function using simple linear algebra. It becomes a convex optimization problem. Fortunately, there exist many efficient algorithms to solve such problems. The algorithm I have used to solve the elastic net problem is called "Cyclic Coordinate Descent", and is described in later sections.

The problem has already been solved for the R programming environment: there exists an R package called *glmnet* [4] that solves the elastic net problem using coordinate descent. It can support linear regression, logistic and multinomial regression models. In this project, I have replicated its functionality in solving the elastic net problem for linear regression in WEKA. The other models supported by *glmnet* are outside the scope of this project.

I have used the paper [4] as a primary reference, because it explains how the elastic net problem can be solved using coordinate descent. The paper leaves out quite a few important details, and a few mathematical derivations were needed to develop a working model. These derivations are included in the appendix, and I have referred to them in later sections.

Chapter 3

Background

One of the popular methods to obtain a simple model is to choose a subset of predictors which best explains the data, among the available predictors. There are many different ways to choose the best set of predictors among the available ones. This approach is usually called "attribute selection" or "feature selection" in the literature.

One of the simpler attribute selection algorithms is called "Forward selection". It is described in [1]. In this approach, we first choose a model building strategy (usually the one we intend to use after attribute selection). We also choose an error function. The algorithm can be summarized as:

1. Start at zero predictors chosen. Build the model and calculate its error.
2. For each predictor $j \in \{1, 2, ...p\}$, build a model by including the predictor and evaluate its error.
3. Choose j to minimize the error. Update the error.
4. Stop if the error was not updated by Step 3. Else, go back to Step 2.

The main disadvantage of Forward Selection is that it does not consider the effect of combinations of attributes in the model. A similar attribute selection algorithm which somewhat mitigates this problem is called "Backward selection". It is also described in [1]. The algorithm can be summarized as:

1. Start at all the predictors chosen. Build the model and calculate its error.
2. For each predictor $j \in \{1, 2, ...p\}$, build a model by excluding the predictor and evaluate its error.
3. Choose j to minimize the error. Update the error, if it changes.
4. Stop if the error was not updated by Step 3. Else, go back to Step 2.

As we can see, Backward Selection does consider the effect of combination of attributes. However, the main disadvantage is that attribute de-selection can be somewhat arbitrary. This problem occurs mainly when there are a few good predictors and many bad ones.

A third attribute selection algorithm is mentioned in [1], called "Stepwise multiple regression". It is similar to forward selection, and the main difference is that selected attributes can be removed later if they are deemed not important enough. The algorithm can be summarized as follows:

1. Start at zero predictors chosen. Build the model and calculate its error.
2. For each predictor $j \in \{1, 2, ...p\}$, build a model by including the predictor and evaluate its error.
3. Choose j to minimize the error. Update the error, if it changes.
4. Stop if the error was not updated by Step 3.
5. Store the reduction in error due to including predictor j.
6. For each predictor $j \in \{1, 2, ...p\}$, build a model by excluding the predictor and evaluate its error.
7. For each predictor $j \in \{1, 2, ...p\}$, calculate the increase in error due to excluding it.
8. Remove predictor j if the error increased by excluding it is smaller than the error reduction originally obtained by including it.
9. Go to Step 2.

A fourth attribute selection algorithm mentioned in [1] is called "Multiple Linear Regression" (MLR). It is very simple:

1. For all combination of attributes, build a model and evaluate its error.
2. Choose the attribute subset that gives the lowest prediction error.

It is computationally expensive because it explores all possible subsets. If the number of attributes is p, then the time needed is $O(2^p)$. Hence, this can only be done of the number of attributes is low. An additional disadvantage is that this algorithm is likely to overfit if there are only a few examples.

Another popular attribute selection algorithm described in the literature is "The forward stagewise algorithm". It is mainly used for linear regression. It first appeared in [5]. In [6], the forward stagewise algorithm is summarized as follows:

1. Center the class and the predictors.

2. Start with $r = y - x\beta$, where β is initially a zero vector, $[xy]$ is the training matrix, and r is the residual vector.

3. Find the predictor x_j most correlated with r. Stop if none of the predictors have any correlation with r.

4. Update $\beta_j = \beta_j + \delta_j$, where $\delta_j = \epsilon \cdot sign[corr(r, x_j)]$;

5. Update $r = r - \delta_j x_j$.

6. Go to Step 3.

As we can see, the algorithm explores the coefficients by making very small jumps. It thus takes a long time to converge.

Another popular algorithm is called Least Angle Regression (LARS). It is an attribute selection algorithm that first appeared in [3]. It is similar to the forward stagewise algorithm described above, with a few major differences. In [6], the LARS algorithm is summarized as follows:

1. Standardize the predictors to have 0 mean and unit variance, and center the class.

2. Start with $r = y - x\beta$, where β is initially a zero vector, $[xy]$ is the training matrix, and r is the residual vector.

3. Find the predictor x_j most correlated with the residual.

4. Start moving β_j from 0 towards its least-squares coefficient $x_j^T r$, updating r continuously.

5. Stop Step 4 when some other x_k is as correlated with the current residual as $\beta_j x_j$.

6. Start moving (β_j, β_k) towards their joint least squares coefficient of the residual on (x_j, x_k).

7. Stop Step 6 when some other x_m is as correlated with the residual as $(\beta_j x_j + \beta_k x_k)$.

8. Start moving $(\beta_j, \beta_k, \beta_m)$.

9. If a coefficient hits zero, drop it from the active set and recompute the direction.

10. Continue in this way until all p predictors have been entered.

11. After p steps, we arrive at the solution.

The LARS algorithm is said to be very efficient, because it has the same time complexity as a full least squares fit using p predictors.

Apart from the above attribute subset selection algorithms, various heuristics for shrinking the coefficients have been suggested in the literature. The LASSO and ridge penalties [5]

have already been discussed in the introduction. LASSO corresponds to an L1 penalty on the coefficients, and ridge regression corresponds to an L2 penalty.

Another coefficient shrinkage method that is quite popular is called the "Non Negative Garotte" (NNG) [2]. Instead of the total squared error, the NNG minimizes the following objective function with respect to the vector c:

$F_{objective} = \sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j c_j)^2$

Here, β_j for $j \in \{1, 2, ...p\}$ are the ordinary least square estimates.
$F_{objective}$ is minimized under the constraint that $c_j \geq 0$ for $j \in \{1, 2, ...p\}$, and $\sum_{j=1}^{p} c_j \leq s$
Obviously, $\beta^{pred} = \beta^T c$ is the coefficient vector used for predictions.
Attribute selection in NNG is performed by tuning the parameter s, and lower values of s tighten the garotte and reduce the number of nonzero coefficients in the model.

An important property of the NNG is that the solution path (for possible values of s) is almost guaranteed to contain a solution that reflects the true importance of each predictor [8]. In other words, the solution path is almost guaranteed to contain the process that actually generated the data.

Finally, it should be noted that using the training error is inappropriate for some of these methods due to a tendency to overfit. In such cases, we need an error estimate for generalization error (such as the cross-validation error), or some other model selection criterion.

Chapter 4

Analysis

As discussed in the introduction, the aim is to solve the elastic net problem for linear regression using Cyclic Coordinate Descent. This translates to minimizing the following objective function over the vector β :

$$F_{objective} = \frac{1}{2N}\sum_{i=1}^{n}(y_i - \beta_0 - x_i^T\beta)^2 + \lambda\sum_{j=1}^{p}(\frac{1}{2}(1-\alpha)\beta_j^2 + \alpha|\beta_j|)$$

Here x_i is the i^{th} instance, y_i is the i^{th} class, β_0 is an intercept term, β is the coefficient vector, N is the number of examples, p is the number of predictors, and the parameters λ, α are subject to $\lambda \geq 0$ and $0 \leq \alpha \leq 1$
This formula was provided in [4]. Before performing our analysis, the following two conditions must be satisfied:

1. All the predictors are centered and have 0 mean.
2. The class has 0 mean and unit variance.

If these conditions are satisfied, we can assume that the intercept term $\beta_0 = 0$. We can then write the objective function as follows:

$$F_{objective} = \frac{1}{2N}\sum_{i=1}^{n}(y_i - x_i^T\beta)^2 + \lambda\sum_{j=1}^{p}(\frac{1}{2}(1-\alpha)\beta_j^2 + \alpha|\beta_j|)$$

4.1 Cyclic Coordinate Descent

The algorithm aims to minimize an objective function F over variables β_j for $j \in \{1, 2, ...p\}$. We must define an update function, $Update()$, such that $Update(j)$ will update and return β_j. We also need a set of variables C_j for $j \in \{1, 2, ...p\}$ to track changes in F due to updating β_j. Next, we must define a threshold to force the algorithm to converge. Let us call it $thresh$. Finally, we must decide the maximum iterations to be allowed. Let us call it max.

The algorithm for Cyclic Coordinate Descent is as follows:

1. Initialize β_j, C_j to 0 for $j \in \{1, 2, ...p\}$, and initialize $iter = 0$.

2. For $j \in \{1, 2, ...p\}$ {
   $\beta_j^{old} = \beta_j$
   $\beta_j = Update(j)$
   $C_j = F(\beta_j) - F(\beta_j^{old})$
   }

3. $iter = iter + 1$

4. If $C_j \geq thresh$ for at least one $j \in \{1, 2, ...p\}$, and if $iter < max$, go to Step 2.

5. Stop.

4.2 The Update() method

The $Update()$ method can be called for any coefficient β_j for $j \in \{1, 2, ...p\}$.
It chooses an optimal value for β_j by minimizing the objective in terms of β_j.
The coefficient update formula mentioned in [4] is as follows:

$$\beta_j^{optimal} = \frac{S(\frac{1}{N}\sum_{i=1}^{n} r_i^{-j} x_{ij}, \lambda\alpha)}{1 + \lambda(1-\alpha)},$$

Here, N is the number of examples, and r_i^{-j} is the i^{th} residual obtained by setting $\beta_j = 0$
Also, $S(p, q)$ with $q \geq 0$ is the Soft Thresholding operator defined by:
$S(p, q) = p - q$ iff $p > q$
$S(p, q) = p + q$ iff $p < -q$
$S(p, q) = 0$ iff $-q \leq p \leq q$

The above formula for $\beta_j^{optimal}$ was derived under the following assumptions:

1. The instances all have equal weights.

2. The predictors have been standardized to have unit variance.

This means that we cannot use the update formula for weighted instance datasets. We also need to standardize the datasets first.
In order to support instance weights, the objective function to be minimized should be:

$$F_{objective} = \frac{1}{2}\frac{\sum_{i=1}^{n} w_i (y_i - x_i^T \beta)^2}{\sum_{i=1}^{n} w_i} + \lambda \sum_{j=1}^{p} (\tfrac{1}{2}(1-\alpha)\beta_j^2 + \alpha|\beta_j|)$$

Here $w_i > 0$ for $i \in \{1, 2, ...n\}$, represent the instance weights. I have derived the optimal update formula for the above objective in Appendix B. The derivation does not make the two assumptions made in the paper. However, the two conditions mentioned at the beginning of this chapter must be satisfied. The derivation in Appendix B makes use of Appendix A, so it is better to read that first.

The generalized update formula is shown to be:

$$\beta_j^{optimal} = \frac{S(\sum_{i=1}^{n} w_i r_i^{-j} x_{ij}, \lambda\alpha \sum_{i=1}^{n} w_i)}{\sum_{i=1}^{n} w_i x_{ij}^2 + \lambda(1-\alpha)\sum_{i=1}^{n} w_i}$$

Clearly, $\sum_{i=1}^{n} w_i x_{ij}^2$ can be precomputed for each $j \in \{1, 2, ...p\}$
Moreover the soft threshold $\lambda\alpha \sum_{i=1}^{n} w_i$ can be precomputed as well.
The denominator $= \sum_{i=1}^{n} w_i x_{ij}^2 + \lambda(1-\alpha)\sum_{i=1}^{n} w_i$, can be precomputed for each $j \in \{1, 2, ...p\}$
So the only term that needs to be computed is $\sum_{i=1}^{n} w_i r_i^{-j} x_{ij}$
In the simplified update formula given in the paper, the corresponding term is $\sum_{i=1}^{n} r_i^{-j} x_{ij}$

Consequently, computing the generalized update formula has the same time complexity as the update formula given in the paper. It is only slightly slower because of the extra computation due to instance weights. Hence, I have used my generalized formula in all cases, including datasets where instance weights do not matter, because this simplifies the code. Note that the authors of the package *glmnet* have used case by case formulas.

4.3 Nature of the coefficient shrinkage

The above discussion shows that we need to compute the term $F_j = \sum_{i=1}^{n} w_i r_i^{-j} x_{ij}$ and then soft threshold F_j with $T = \lambda\alpha \sum_{i=1}^{n} w_i$. Based on the generalized formula, a coefficient update will be zero iff $|F_j| \leq T$.

The term F_j can be described as: "Dot product of the j^{th} predictor with the residual obtained by excluding the j^{th} coefficient". Since the magnitude of a dot product is proportional to the correlation coefficient, it is intuitively obvious that F_j is needed to calculate the coefficient of the j^{th} predictor.

In the case of ridge regression, with an L2 norm penalty, we see that the soft threshold $T = 0$ since $\alpha = 0$. Obviously, the coefficient β_j will be shrunk to 0 if and only if $|F_j| \leq 0$, which means that the dot product $F_j = 0$. Clearly, this is very unlikely to happen. Thus we can say that, with probability approaching 1, all the predictors in ridge regression will have nonzero coefficients.

As α is increased from 0 to 1, the value of T increases and the inequality $|F_j| \leq T$ relaxes. So a dot product F_j during an update is more likely to satisfy the inequality. Thus we can say that, as we move from the ridge to the lasso penalty, it becomes more and more likely for predictors to be dropped from the model (by getting 0 as their coefficient).

For LASSO and elastic net models, the same effect is achieved when λ is increased. This is because increasing the value of λ would also increase T and make it more likely for predictors to be dropped from the model.

To summarize, ridge regression models will have nonzero coefficients for each predictor, LASSO models will have the least number of nonzero coefficients, and elastic net models will be between the ridge and LASSO models. For all penalized models (LASSO, ridge, elastic net), increasing the value of λ leads to lower values of the coefficients.

4.4 Computing the solution path

Based on the discussion in the preceding sections, we know that we can compute the coefficient vector β by choosing a specific α and λ and performing Coordinate Descent. We can visualize β as a point in the p dimensional coefficient-space.

We note that α tunes the nature of the penalty term, while λ tunes the magnitude. Practically speaking, we may be able to decide the nature of the penalty, but we would not know the ideal magnitude. For this reason, we may wish to see the coefficient vectors obtained for various values of λ (by keeping α constant).

The set of coefficient vectors obtained for a sequence of λ values is called a "solution path" in the literature. It corresponds to a set of points in the p dimensional coefficient-space. From the solution path, we can select a model that suits our requirements.

The package *glmnet* computes its own λ sequence by default. This feature can be overridden if the user supplies a λ sequence. The default λ sequence is computed by first choosing a suitably large value of λ, such that the coefficient vector β is guaranteed to be very close to the origin. Let us call this λ_{zero}. The other λ values are chosen in logarithmic decrements from λ_{zero}.

Recall that the j^{th} coefficient will be updated to 0 iff $|F_j| \leq T$. We also know that initially, all the coefficients $\beta_j = 0$ for $j \in \{1, 2, ...p\}$. Now, if we can guarantee that all coefficients are updated to 0 in the first iteration, then we can say that the coefficients will always remain 0. This means that the resulting coefficient vector β will be at the origin. This is called the zero model (or the intercept model).
Now, let F_j^{init} = Initial value of $F_j = \sum_{i=1}^{n} w_i r_i^{-j} x_{ij}$
We know that initially, $r_i^{-j} = y_i$ because all the coefficients are 0. Hence, $F_j^{init} = \sum_{i=1}^{n} w_i y_i x_{ij}$
To ensure that the output is a zero model, we must satisfy the below inequality:

$$max(|F_1^{init}|, |F_2^{init}|, ...|F_p^{init}|) \leq T$$

Clearly, this inequality will be satisfied for very high values of λ, since $T = \lambda\alpha \sum_{i=1}^{n} w_i$
This also makes intuitive sense: if the penalty term is too high, then the best model will be the zero model. Now, if we want to find the minimum λ such that our model is guaranteed to be a zero model, we must use the following equation:

$$max(|F_1^{init}|, |F_2^{init}|, ...|F_p^{init}|) = T = \lambda\alpha \sum_{i=1}^{n} w_i. \text{ So,}$$

$$\lambda_{zero} = \frac{max(|F_1^{init}|, |F_2^{init}|, ...|F_p^{init}|)}{\alpha \sum_{i=1}^{n} w_i}$$

This formula only works for $\alpha > 0$. If $\alpha = 0$, then there exists no λ for which we are guaranteed to get a zero model. The best we can do in this case is choose some large value for λ_{zero} and hope that the resulting coefficient vector will be close to the origin.

Now, after calculating λ_{zero}, we can calculate some $\lambda_{min} = \epsilon \cdot \lambda_{zero}$, where ϵ is a pre-defined constant close to 0. With λ_{zero} and λ_{min}, we can build a list of *len* values of λ, logarithmically descending from λ_{zero} to λ_{min}. Here, *len* is the pre-defined list length. Let us call this list L.

The algorithm to build L can be summarized as follows :

1. If an user-supplied λ sequence is available, fill L in descending order and stop.
2. If $\alpha > 0$, initialize $\lambda_{zero} = \dfrac{max(|F_1^{init}|, |F_2^{init}|, ...|F_p^{init}|)}{\alpha \sum_{i=1}^{n} w_i}$
3. If $\alpha = 0$, initialize λ_{zero} to some arbitrary large value.
4. Initialize $\lambda_{min} = \epsilon \cdot \lambda_{zero}$
5. Initialize the ratio $x = (\frac{\lambda_{min}}{\lambda_{zero}})^{\frac{1}{len-1}}$
6. Initialize the list L of length len
7. For each $i \in \{1, 2, ...len\}$, set $L[i] = \lambda_{zero} \cdot x^{i-1}$
8. Stop. We have built the list L in descending order.

In [4], the algorithm to compute the solution path from the list L is called "Pathwise Coordinate Descent". It has been implemented as a standalone function in the package *glmnet*.

The algorithm for Pathwise Coordinate Descent is as follows:

1. Initialize the coefficient vector β such that $\beta_j = 0$ for $j \in \{1, 2, ...p\}$
2. For $m \in \{1, 2, ...length(L)\}$ {
   $\lambda_{current} = L[m]$
   $CyclicCoordinateDescent(\beta, \lambda_{current})$
   $Save(\beta)$ as the m^{th} solution.
   }
3. Stop. We have saved the solutions corresponding to all values of λ present in L.

Hence, we can study the solution path for a constant α, over a sequence of λ values. This algorithm is much faster than individually running $CyclicCoordinateDescent$ on each λ in the list L. It is easy to see why: In every iteration, the vector β initially has the coefficients for the previous λ in L. This effectively gives us a "warm start".

4.5 Cross Validated parameter selection

As noted in the preceding section, we may know the nature of the penalty to impose (determined by α) but not the magnitude (determined by λ). We may perform Pathwise Coordinate Descent over a list of λ values, and choose the ideal solution from the solution path.

The authors of *glmnet* use k fold cross validation to compare the different solutions. The cross validation error is a good estimate of the generalization error (i.e, how well a model can

generalize over "unseen" data). Cross validated parameter selection is the main algorithm of the WEKA implementation of *glmnet*, and it makes use of all the algorithms discussed in the preceding sections.

The algorithm for cross validated parameter selection is as follows :

1. Build the list L of λ values using the algorithm described in the preceding section.
2. Randomly shuffle the dataset D.
3. Split D into k equally sized subsets (or "folds") called $F_1, F_2, ...F_k$
4. Initialize an array A of size $length(L)$. Fill A with zeroes.
5. For i $\in \{1, 2, ...k\}$ {
   Initialize the training dataset as $TrainSet = D - F_i$
   Initialize the test dataset as $TestSet = F_i$
   Run Pathwise Coordinate Descent over $TrainSet$ to compute the solution path.
   Test each solution (i.e coefficient vector) from the solution path against $TestSet$.
   Calculate the Mean Squared Error (MSE) for each solution against $TestSet$.
   For each $m \in \{1, 2, ...length(L)\}$, add the MSE for the m^{th} solution to $A[m]$.
   }
6. Divide all values in A by k. $A[m]$ is now the average error of the m^{th} solution over k folds
7. Run Pathwise Coordinate Descent over the full dataset D to compute the solution path.
8. Based on the error values in A, choose the best solution from the solution path.
9. Stop.

The most common way to choose the best solution in Step 8 is to choose the index of the minimum error in A. This corresponds to the λ value that has the lowest generalization error. If all the predictors are relevant, then this approach is likely to choose a solution closest to the least squares solution, and this is likely to be the solution furthest away from the origin. The WEKA implementation of *glmnet* uses this approach by default.

If we wish to obtain a more parsimonious solution, we need to select a solution closer to the origin without sacrificing too much predictive accuracy. Recall that the list of λ values is sorted in descending order, and that $A[1]$ corresponds to λ_{zero}. This means that a lower index will correspond to a solution closer to the origin. A useful heuristic that is employed in the package *glmnet* is to choose the lowest index of A whose error is within 1 Standard Error of the minimum. This approach is also supported in the WEKA implementation of *glmnet*.

Chapter 5

Design and Implementation

The main design objectives of the WEKA implementation of the elastic-net method are:

1. Its speed and accuracy should be comparable with *glmnet* (written in FORTRAN).
2. It should be able to handle instance weights or unstandardized data.
3. It should achieve speedups when the training data is sparse.

The major difference between my design and that of *glmnet* is that I have used the generalized update formula in all cases, including datasets where instance weights do not matter. The previous chapter has explained how weighted instances are handled in this project. The authors of the package *glmnet* use different formulas for various cases such as: weighted instances, sparse input data, and unstandardized input data.
To recap, the update formula used is:

$$\beta_j^{optimal} = \frac{S(\sum_{i=1}^{n} w_i r_i^{-j} x_{ij}, \lambda\alpha \sum_{i=1}^{n} w_i)}{\sum_{i=1}^{n} w_i x_{ij}^2 + \lambda(1-\alpha)\sum_{i=1}^{n} w_i},$$

where everything except $F_j = \sum_{i=1}^{n} w_i r_i^{-j} x_{ij}$ can be precomputed.
Clearly, F_j is the most expensive computation here. The paper [4] suggests two ways of computing it: The "Naive" and "Covariance" update methods. Essentially, they are based on two different ways of rewriting F_j, however the actual formulas used in this project differ from [4] because of the different update formula. (Refer Appendix C). The WEKA implementation of *ElasticNet* uses "Covariance updates" by default.

5.1 Method 1: Naive Updates

We rewrite F_j as $F_j = \sum_{i=1}^{n} w_i x_{ij} r_i + \beta_j \sum_{i=1}^{n} w_i x_{ij}^2$ (Refer Appendix C),

where β_j is the current coefficient, and the other variables have their usual meanings.
From this, it is clear that we must store a list of the N residuals - one for each instance.

The initial value of a residual r_i is y_i, since all the coefficients are initially 0. For each coefficient update, we must loop over the all the instances and compute the sum $\sum_{i=1}^{n} w_i x_{ij} r_i$. This is clearly an $O(N)$ computation.

After that, it's a simple $O(1)$ computation to find F_j, and then another $O(1)$ computation to find the new coefficient using F_j. After finding the new coefficient, we must again loop over all the instances and update each residual. We can update a residual r_i by adding the term $((\beta_j^{old} - \beta_j^{new}) \cdot x_{ij})$ to it. Clearly, updating all the residuals is an $O(N)$ computation.

To check the termination condition, we need to measure the change in the objective function. It is fairly easy to measure the change in the penalty term because it depends only on β_j. To update the penalty term, we add:

$\frac{1}{2}\lambda(1-\alpha) \cdot (\beta_j^{new^2} - \beta_j^{old^2}) + \lambda\alpha \cdot (|\beta_j^{new}| - |\beta_j^{old}|)$

This is a fairly straightforward $O(1)$ computation.
For the squared loss term, we can maintain the weighted sum of squared residuals:
$S_2 = \sum_{i=1}^{n} w_i r_i^2$. We need to update this term each time a residual changes. This introduces some computational overhead, but the time complexity remains the same. After a coefficient update, the change in the squared loss is obviously $(S_2^{new} - S_2^{old})$.
Overall, a single coefficient update takes $O(N)$ time where N is the number of examples.

5.2 Method 2: Covariance Updates

From Appendix C, we have $F_j = \sum_{i=1}^{n} w_i x_{ij} y_i - \sum_{k=1}^{p} (\beta_k \sum_{i=1}^{n} w_i x_{ij} x_{ik}) + \beta_j \sum_{i=1}^{n} w_i x_{ij}^2$

We also know that $G_j = \sum_{i=1}^{n} w_i x_{ij} r_i = \sum_{i=1}^{n} w_i x_{ij} y_i - \sum_{k=1}^{p} (\beta_k \sum_{i=1}^{n} w_i x_{ij} x_{ik})$

We need to store and update all G_k for $k \in \{1, 2, ...p\}$.

Since the coefficient vector β is initially 0, the initial value for each G_k would be $\sum_{i=1}^{n} w_i x_{ik} y_i$. Finally, we must initialize a "covariance matrix" M, of dimensions $p \times p$. The matrix is meant to satisfy $M[a, b] = \sum_{i=1}^{n} w_i x_{ia} x_{ib}$. That is, $M[a, b]$ must store the dot product of the a^{th} and b^{th} predictors. The matrix is initialized with zeroes, and is filled "on demand".

To update the coefficient β_j, we must calculate F_j using G_j (which is an $O(1)$ computation), and then calculate the new coefficient using F_j (also $O(1)$). After updating β_j, we must loop over all the predictors and update all G_k for $k \in \{1, 2, ...p\}$. This is an $O(p)$ computation because each G_k can be updated in $O(1)$ time.

For a specific k, G_k can be updated by adding the term $((\beta_j^{old} - \beta_j^{new}) \cdot M[j, k])$ to it. Thus, we need the j^{th} row of matrix M to update all G_k for $k \in \{1, 2, ...p\}$. We do not need to fill the entire matrix at the start - we only need to fill the j^{th} row if the j^{th} coefficient changes. This approach has an obvious advantage if many of the coefficients remain 0 till the end.

Moreover, we can take advantage of the fact that for any $a, b \in \{1, 2, ...p\}$, $M[a, b] = M[b, a]$. This is because the dot product is commutative. So whenever we need to fill $M[a, b]$, we can just copy the value of $M[b, a]$ if that is already filled.

Overall, the time complexity of a coefficient update is $O(p)$, if the j^{th} row of matrix M has already been filled. Otherwise, it is $O(Np)$, because we would need to compute the dot product of each predictor with predictor j.

Note that each iteration of Cyclic Coordinate Descent has a chance of filling one or more rows of the matrix M. On average, each iteration will be slightly faster than the previous iteration because the matrix M will be fuller. The algorithm will eventually attain peak performance after it chooses an "active set" of nonzero coefficients to modify in successive iterations.

To check the termination condition, we need to measure the change in the objective function. As mentioned earlier, it is fairly easy to measure the change in the penalty term. However, measuring the change in the squared loss term is much harder since we do not store the residuals. In Appendix D, I have derived a formula for updating the squared loss.
The term to be added to update the squared loss is:

$$F_{error}^{new} - F_{error}^{old} = (\beta_j^{new^2} - \beta_j^{old^2}) \cdot \frac{\sum_{i=1}^{n} w_i x_{ij}^2}{2\sum_{i=1}^{n} w_i} - (\beta_j{}^{new} - \beta_j{}^{old}) \cdot \frac{1}{\sum_{i=1}^{n} w_i}(G_j^{new} + \beta_j^{new} \sum_{i=1}^{n} w_i x_{ij}^2)$$

The formula uses G_j, and fortunately, we already store and update all G_k for $k \in \{1, 2, ...p\}$. Note that we can only apply this formula after updating G_j (the updated value is referred to as G_j^{new}). The time complexity for this formula is clearly $O(1)$. Hence, updating the objective function does not change the overall time complexity.

5.3 Sparse Datasets

A dataset is called "sparse" if the majority of the [x y] training matrix is 0. Many Machine Learning algorithms take advantage of this sparsity in order to achieve speedups. However, the authors of the paper [4], have not mentioned how to take advantage of sparsity for Pathwise Coordinate Descent.

The biggest obstacle to taking advantage of sparsity is that the coefficient update formula only works if all the predictors are centered. This is one of the conditions mentioned at the beginning of Chapter 4. However, centering a predictor shifts all values in the column by the column mean: All sparsity is now lost.

The solution is to incorporate centering within the algorithm itself, rather than centering the data at the beginning. Fortunately, this is fairly easy to do since most of the computations are rather simple: Appendix E shows how we can incorporate centering in some of the computations we need to perform. It turns out that these changes do not increase the time complexity - they only increase the precomputation to be performed.

So now, we can take advantage of the input data sparsity while computing various dot products: obviously, we skip the indices where one, or both variables are known to be 0. Another, less obvious way to take advantage of sparsity is in storing residuals (for Method 1).

Recall that in Method 1, after updating a coefficient, we iterate over all the instances to update each residual by adding the term $((\beta_j^{old} - \beta_j^{new}) \cdot x_{ij})$ to it. Now, if an $x_{ij} = 0$, we can skip that instance. However, this will never happen, because x_{ij} is implicitly centered.

So, I have defined an "uncentered residual", $r^{'}$, such that:

$r_i^{'} = y_i - \sum_{j=1}^{p} \beta_j x_{ij}^{uncentered}$, where $r_i^{'}$ is the i^{th} "uncentered residual".

These "uncentered residuals" are used for all the computations in Method 1.
Now, after updating a coefficient β_j, we only need to update a residual $r_i^{'}$ if $x_{ij} \neq 0$, for $i \in \{1, 2, ...n\}$. The $O(1)$ changes needed to incorporate "uncentered residuals" in our computations are explained in Appendix F. They are:

1. Store and update the term $Q = \sum_{j=1}^{p} \beta_j \mu_j$, whenever a coefficient changes.
2. Store and update $S_1 = \sum_{i=1}^{n} w_i r_i^{'}$, whenever a residual changes.
3. Store and update $S_2 = \sum_{i=1}^{n} w_i r_i^{'2}$, whenever a residual changes.
4. Use $G_j = \sum_{i=1}^{n} w_i r_i x_{ij}^{centered} = \sum_{i=1}^{n} w_i x_{ij} r_i^{'} - \mu_j \cdot S_1$
5. After updating Q, S_1, S_2 we can find the squared loss using this formula:
   $F_{error} = \frac{1}{2\sum_{i=1}^{n} w_i}(S_2 + 2QS_1) + \frac{Q^2}{2}$

5.4 Making predictions

The basic formula for predicting the class from an instance vector x and coefficient vector β is:

$y^{pred} = \sum_{j=1}^{p} \beta_j x_j$

This formula can be used directly only if the conditions mentioned in Chapter 4 are satisfied. If not, we will have implicitly centered the predictors or standardized the class while training our model (Refer Appendix E), and these must be reversed while making predictions.
For example, if we have implicitly centered our predictors, the prediction formula must be:

$y^{pred} = \sum_{j=1}^{p} \beta_j (x_j - \mu_j)$, where μ_j is the mean of predictor j.

And if we have implicitly standardized the class, the prediction formula must be:

$y^{pred} = \mu_{class} + \sigma_{class} \cdot \sum_{j=1}^{p} \beta_j x_j$

Here, μ_{class} is the class mean and σ_{class} is the standard deviation of the class.

Obviously, if we have implicitly done both of the above, the prediction formula must be:

$y^{pred} = \mu_{class} + \sigma_{class} \cdot \sum_{j=1}^{p} \beta_j (x_j - \mu_j)$

Finally, we can write $\sum_{j=1}^{p} \beta_j (x_j - \mu_j) = \sum_{j=1}^{p} \beta_j x_j - \sum_{j=1}^{p} \beta_j \mu_j$

We can precompute the term $Q = \sum_{j=1}^{p} \beta_j \mu_j$, as this would reduce the computational overhead.

Chapter 6

Experiments and Results

Several experiments were conducted with the WEKA implementation of elastic net regression, called $weka.functions.ElasticNet$. Primarily, the predictive performance of $ElasticNet$ is compared with the R implementation, called *glmnet* [4]. It is possible to integrate WEKA with the R environment by installing the "RPlugin" package. This would install a WEKA classifier called $weka.mlr.MLRClassifier$, which can run R algorithms like *glmnet* from within WEKA.

As mentioned earlier, *glmnet* can support various models, such as linear regression, logistic and multinomial regression. The default model is linear regression, and this is what is used throughout, because the WEKA implementation only solves the elastic net problem for linear regression. The tunable parameters of *glmnet* are:

1. α, which tunes the nature of the elastic net penalty (default value 1).
2. $thresh$, which tunes the convergence threshold (default value 1e-7).
3. $nfolds$, which is the number of folds to be used for internal cross validation (default 10).
4. $stdandardize$, which determines whether the data is to be standardized first (Boolean).

The $ElasticNet$ classifier has parameters 1,2 and 3 with the same default values. The parameter $stdandardize$ is set to FALSE so that both implementations deal with unstandardized data. In addition, the R implementation has two hidden parameters which can be obtained by executing *glmnet* in the R environment and carefully observing the results. The parameters are:

1. $numModels = 100$. This is the length of the default lambda sequence (to be generated).
2. $\epsilon = 10^{-4}$. This value can be obtained by dividing λ_{max} by λ_{min}.

The $ElasticNet$ classifier also has these parameters, but they are tunable. The experiments were conducted for various values of α over various regression datasets (with numeric attributes and no missing values). The experiments were run from WEKA's Experimenter interface. Each dataset was tested by 10 fold cross validation with 5 repetitions. Each experiment measures the correlation coefficients of the two classifiers over 37 datasets, and performs a statistical significance test, using the paired corrected resampled t-test [7]. The results are shown in Tables 6.1 to 6.6.

Table 6.1: Comparison of *glmnet* and *ElasticNet* for $\alpha = 1$

Datasets (Numeric attributes and class)	(1)	(2)
2dplanes	0.84±0.00	0.84±0.00
Brix2-Full	0.76±0.03	0.76±0.03
ailerons	0.00±0.00	0.90±0.01 ∘
autoPrice.names	0.89±0.06	0.88±0.07
bank32nh	0.70±0.02	0.73±0.02 ∘
bank8FM	0.96±0.00	0.97±0.00 ∘
baskball	0.58±0.19	0.60±0.19
bodyfat.names	0.97±0.02	0.97±0.02
bolts	0.80±0.36	0.84±0.33
cal-housing	0.79±0.01	0.79±0.01
cpu	0.89±0.10	0.92±0.06
cpu-act	0.70±0.02	0.78±0.02 ∘
cpu-edited	0.89±0.10	0.92±0.06
cpu-small	0.70±0.02	0.76±0.02 ∘
delta-ailerons	0.00±0.00	0.82±0.01 ∘
delta-elevators	0.00±0.00	0.79±0.01 ∘
detroit	0.14±0.53	0.18±0.52
diabetes-numeric	0.48±0.48	0.48±0.47
elevators	0.14±0.02	0.45±0.02 ∘
fried	0.85±0.00	0.85±0.00
gascons	0.97±0.15	0.97±0.15
house-16H	0.11±0.04	0.11±0.04
house-8L.arff	0.11±0.04	0.44±0.04 ∘
kin8nm	0.64±0.02	0.64±0.02 ∘
longley	0.56±0.54	0.56±0.54
machine-cpu	0.89±0.10	0.92±0.06
pol	0.68±0.01	0.68±0.01
pollution	0.79±0.17	0.79±0.19
puma32H	0.47±0.03	0.47±0.03
puma8NH	0.61±0.02	0.61±0.02
pwLinear	0.87±0.06	0.87±0.06
pyrim	0.50±0.40	0.66±0.27
quake	0.06±0.07	0.06±0.07
stock	0.93±0.02	0.93±0.02
triazines	0.25±0.15	0.44±0.19
vineyard	0.70±0.30	0.70±0.30
wisconsin	0.37±0.19	0.38±0.19
Average	0.58	0.69

∘, • statistically significant improvement or degradation

(1) mlr.MLRClassifier '-learner regr.glmnet -params "standardize=FALSE, alpha=1"'
(2) functions.ElasticNet '-alpha 1'

Table 6.2: Comparison of *glmnet* and *ElasticNet* for $\alpha = 0.8$

Datasets (Numeric attributes and class)	(1)	(2)
2dplanes	0.84±0.00	0.84±0.00
Brix2-Full	0.76±0.03	0.76±0.03
ailerons	0.00±0.01	0.90±0.01 ∘
autoPrice.names	0.89±0.06	0.88±0.07
bank32nh	0.70±0.02	0.73±0.02 ∘
bank8FM	0.96±0.00	0.97±0.00 ∘
baskball	0.58±0.19	0.60±0.20
bodyfat.names	0.88±0.04	0.88±0.04
bolts	0.80±0.36	0.84±0.33
cal-housing	0.79±0.01	0.79±0.01
cpu	0.89±0.10	0.92±0.06
cpu-act	0.70±0.02	0.78±0.02 ∘
cpu-edited	0.89±0.10	0.92±0.06
cpu-small	0.70±0.02	0.76±0.02 ∘
delta-ailerons	0.00±0.00	0.79±0.01 ∘
delta-elevators	0.00±0.00	0.79±0.01 ∘
detroit	0.14±0.53	0.18±0.52
diabetes-numeric	0.48±0.48	0.48±0.48
elevators	0.30±0.02	0.43±0.02 ∘
fried	0.85±0.00	0.85±0.00
gascons	0.97±0.15	0.97±0.15
house-16H	0.11±0.04	0.11±0.04
house-8L.arff	0.11±0.04	0.44±0.05 ∘
kin8nm	0.64±0.02	0.64±0.02
longley	0.56±0.54	0.56±0.54
machine-cpu	0.89±0.10	0.92±0.06
pol	0.68±0.01	0.68±0.01
pollution	0.79±0.17	0.79±0.19
puma32H	0.47±0.03	0.47±0.03
puma8NH	0.61±0.02	0.61±0.02
pwLinear	0.87±0.06	0.87±0.06
pyrim	0.58±0.40	0.67±0.27
quake	0.06±0.07	0.06±0.07
stock	0.93±0.02	0.93±0.02
triazines	0.24±0.15	0.44±0.19
vineyard	0.70±0.30	0.70±0.29
wisconsin	0.37±0.19	0.38±0.20
Average	0.59	0.68

∘, • statistically significant improvement or degradation

(1) mlr.MLRClassifier '-learner regr.glmnet -params "standardize=FALSE, alpha=0.8"'
(2) functions.ElasticNet '-alpha 0.8'

Table 6.3: Comparison of *glmnet* and *ElasticNet* for $\alpha = 0.6$

Datasets (Numeric attributes and class)	(1)	(2)
2dplanes	0.84±0.00	0.84±0.00
Brix2-Full	0.76±0.03	0.76±0.03
ailerons	0.07±0.03	0.90±0.01 ∘
autoPrice.names	0.89±0.06	0.88±0.06
bank32nh	0.71±0.02	0.73±0.02 ∘
bank8FM	0.96±0.00	0.97±0.00 ∘
baskball	0.58±0.19	0.61±0.20
bodyfat.names	0.86±0.05	0.86±0.05
bolts	0.80±0.36	0.83±0.33
cal-housing	0.79±0.01	0.79±0.01
cpu	0.89±0.10	0.92±0.06
cpu-act	0.70±0.02	0.78±0.02 ∘
cpu-edited	0.89±0.10	0.92±0.06
cpu-small	0.70±0.02	0.76±0.02 ∘
delta-ailerons	0.00±0.00	0.78±0.01 ∘
delta-elevators	0.65±0.01	0.79±0.01 ∘
detroit	0.14±0.53	0.18±0.52
diabetes-numeric	0.48±0.48	0.48±0.47
elevators	0.37±0.02	0.42±0.02 ∘
fried	0.85±0.00	0.85±0.00
gascons	0.97±0.15	0.95±0.20
house-16H	0.11±0.04	0.11±0.04
house-8L.arff	0.11±0.04	0.44±0.05 ∘
kin8nm	0.64±0.02	0.64±0.02
longley	0.56±0.54	0.56±0.54
machine-cpu	0.89±0.10	0.92±0.06
pol	0.68±0.01	0.68±0.01
pollution	0.79±0.17	0.79±0.18
puma32H	0.47±0.03	0.47±0.03
puma8NH	0.61±0.02	0.61±0.02
pwLinear	0.87±0.06	0.87±0.06
pyrim	0.61±0.40	0.69±0.28
quake	0.06±0.07	0.06±0.07
stock	0.93±0.02	0.93±0.02
triazines	0.28±0.16	0.44±0.20
vineyard	0.70±0.30	0.70±0.30
wisconsin	0.37±0.19	0.38±0.20
Average	0.61	0.68

∘, • statistically significant improvement or degradation

(1) mlr.MLRClassifier '-learner regr.glmnet -params "standardize=FALSE, alpha=0.6"'
(2) functions.ElasticNet '-alpha 0.6'

Table 6.4: Comparison of *glmnet* and *ElasticNet* for $\alpha = 0.4$

Datasets (Numeric attributes and class)	(1)	(2)
2dplanes	0.84±0.00	0.84±0.00
Brix2-Full	0.76±0.03	0.76±0.03
ailerons	0.07±0.03	0.90±0.01 ∘
autoPrice.names	0.89±0.06	0.88±0.06
bank32nh	0.71±0.02	0.73±0.02 ∘
bank8FM	0.96±0.00	0.97±0.00 ∘
baskball	0.58±0.19	0.60±0.19
bodyfat.names	0.86±0.05	0.86±0.05
bolts	0.80±0.36	0.84±0.33
cal-housing	0.79±0.01	0.79±0.01
cpu	0.89±0.10	0.92±0.06
cpu-act	0.70±0.02	0.78±0.02 ∘
cpu-edited	0.89±0.10	0.92±0.06
cpu-small	0.70±0.02	0.76±0.02 ∘
delta-ailerons	0.00±0.00	0.78±0.01 ∘
delta-elevators	0.66±0.01	0.79±0.01 ∘
detroit	0.14±0.53	0.18±0.52
diabetes-numeric	0.48±0.48	0.48±0.47
elevators	0.40±0.02	0.42±0.02 ∘
fried	0.85±0.00	0.85±0.00
gascons	0.97±0.15	0.91±0.28
house-16H	0.11±0.04	0.11±0.04
house-8L.arff	0.11±0.04	0.44±0.06 ∘
kin8nm	0.64±0.02	0.64±0.02
longley	0.56±0.54	0.56±0.54
machine-cpu	0.89±0.10	0.92±0.06
pol	0.68±0.01	0.68±0.01
pollution	0.79±0.17	0.78±0.20
puma32H	0.47±0.03	0.47±0.03
puma8NH	0.61±0.02	0.61±0.02
pwLinear	0.87±0.06	0.87±0.06
pyrim	0.62±0.40	0.71±0.27
quake	0.06±0.07	0.06±0.07
stock	0.93±0.02	0.93±0.02
triazines	0.33±0.17	0.44±0.19
vineyard	0.70±0.30	0.70±0.29
wisconsin	0.37±0.19	0.38±0.20
Average	0.61	0.68

∘, • statistically significant improvement or degradation

(1) mlr.MLRClassifier '-learner regr.glmnet -params "standardize=FALSE, alpha=0.4"'
(2) functions.ElasticNet '-alpha 0.4'

Table 6.5: Comparison of *glmnet* and *ElasticNet* for $\alpha = 0.2$

Datasets (Numeric attributes and class)	(1)	(2)
2dplanes	0.84±0.00	0.84±0.00
Brix2-Full	0.76±0.03	0.76±0.03
ailerons	0.07±0.03	0.90±0.01 ∘
autoPrice.names	0.89±0.06	0.89±0.06
bank32nh	0.72±0.02	0.73±0.02 ∘
bank8FM	0.96±0.00	0.97±0.00 ∘
baskball	0.57±0.19	0.60±0.19
bodyfat.names	0.85±0.05	0.85±0.05
bolts	0.80±0.36	0.84±0.33
cal-housing	0.79±0.01	0.79±0.01
cpu	0.89±0.10	0.92±0.06
cpu-act	0.70±0.02	0.78±0.02 ∘
cpu-edited	0.89±0.10	0.92±0.06
cpu-small	0.70±0.02	0.76±0.02 ∘
delta-ailerons	0.00±0.00	0.78±0.01 ∘
delta-elevators	0.66±0.01	0.78±0.01 ∘
detroit	0.14±0.53	0.18±0.52
diabetes-numeric	0.48±0.48	0.48±0.47
elevators	0.41±0.02	0.41±0.02 ∘
fried	0.85±0.00	0.85±0.00
gascons	0.96±0.15	0.96±0.15
house-16H	0.11±0.04	0.11±0.04
house-8L.arff	0.11±0.04	0.42±0.08 ∘
kin8nm	0.64±0.02	0.64±0.02
longley	0.56±0.54	0.56±0.54
machine-cpu	0.89±0.10	0.92±0.06
pol	0.68±0.01	0.68±0.01
pollution	0.79±0.17	0.78±0.18
puma32H	0.47±0.03	0.47±0.03
puma8NH	0.61±0.02	0.61±0.02
pwLinear	0.87±0.06	0.87±0.06
pyrim	0.63±0.39	0.69±0.32
quake	0.06±0.07	0.06±0.07
stock	0.93±0.02	0.93±0.02
triazines	0.40±0.17	0.44±0.19
vineyard	0.70±0.30	0.70±0.30
wisconsin	0.37±0.19	0.38±0.20
Average	0.62	0.68

∘, • statistically significant improvement or degradation

(1) mlr.MLRClassifier '-learner regr.glmnet -params "standardize=FALSE, alpha=0.2"'
(2) functions.ElasticNet '-alpha 0.2'

Table 6.6: Comparison of *glmnet* and *ElasticNet* for $\alpha = 0.001$

Datasets (Numeric attributes and class)	(1)	(2)
2dplanes	0.84±0.00	0.84±0.00
Brix2-Full	0.71±0.03	0.71±0.03
ailerons	0.70±0.01	0.75±0.01 ∘
autoPrice.names	0.89±0.06	0.89±0.06
bank32nh	0.72±0.02	0.72±0.02
bank8FM	0.96±0.00	0.96±0.00
baskball	0.57±0.19	0.58±0.19
bodyfat.names	0.85±0.05	0.85±0.05
bolts	0.84±0.34	0.84±0.33
cal-housing	0.48±0.03	0.48±0.03 •
cpu	0.89±0.10	0.91±0.08
cpu-act	0.70±0.02	0.74±0.02 ∘
cpu-edited	0.89±0.10	0.91±0.08
cpu-small	0.70±0.02	0.73±0.02 ∘
delta-ailerons	0.00±0.00	0.74±0.01 ∘
delta-elevators	0.66±0.01	0.67±0.01 ∘
detroit	0.14±0.53	0.14±0.53
diabetes-numeric	0.48±0.48	0.50±0.46
elevators	0.41±0.02	0.41±0.02
fried	0.85±0.00	0.85±0.00
gascons	0.85±0.39	0.83±0.41
house-16H	0.11±0.04	0.11±0.04
house-8L.arff	0.11±0.04	0.11±0.04
kin8nm	0.64±0.02	0.64±0.02
longley	0.56±0.54	0.56±0.54
machine-cpu	0.89±0.10	0.91±0.08
pol	0.68±0.01	0.68±0.01
pollution	0.79±0.20	0.79±0.20
puma32H	0.47±0.03	0.47±0.03
puma8NH	0.61±0.02	0.61±0.02
pwLinear	0.87±0.06	0.87±0.06
pyrim	0.66±0.36	0.69±0.32
quake	0.06±0.07	0.06±0.07
stock	0.93±0.02	0.93±0.02
triazines	0.43±0.17	0.44±0.18
vineyard	0.68±0.33	0.68±0.33
wisconsin	0.39±0.20	0.39±0.20
Average	0.62	0.65

∘, • statistically significant improvement or degradation

(1) mlr.MLRClassifier '-learner regr.glmnet -params "standardize=FALSE, alpha=1e-3"'
(2) functions.ElasticNet '-alpha 0.001'

6.1 Discussion

It is clear that the *ElasticNet* classifier performs very similar to *glmnet*. For most datasets, we can see that the correlation coefficients are very close. By repeating each run 5 times, we seem to have mitigated the randomness introduced by internal cross validation. However, though the results are mostly similar, there are some notable exceptions. It is easy to spot several cases where the performance of the two algorithms differs significantly. We will now discuss these exceptions, and speculate on the possible causes.

First, consider the entry for the dataset "delta-ailerons.arff" in Table 6.1. Observe that *glmnet* obtained a correlation coefficient close to 0 whereas *ElasticNet* obtained 0.82. However, it is possible to obtain a correlation coefficient of 0.823 by directly running *glmnet* for "delta-ailerons.arff" in the R environment. Note that this result is very close to that obtained by *ElasticNet*. The relevant R code is present in Appendix G.

The only possible explanation for this is that there is a bug in the R/WEKA integration. This phenomenon can be observed whenever *glmnet* seems to obtain a correlation coefficient close to 0 and *ElasticNet* performs significantly better. Fortunately, we can independently verify that most of the results obtained by running *glmnet* within the MLR classifier are very similar to the results obtained by directly running *glmnet* in the R environment. Hence, this bug does not affect the credibility of the other results.

Now, consider the dataset "cpu-act.arff" in Table 6.1. The correlation coefficient obtained by *ElasticNet* is significantly better than that obtained by *glmnet*. Further investigation revealed that the MLR classifier produced the exact same output as *glmnet* executed in the R environment. Clearly, this is not an R/WEKA integration problem.

The discrepancy is due to the fact that in this specific case, *glmnet* appears to truncate its λ sequence after first generating a full sequence. To see why, note that the first 66 values of the sequence generated by *ElasticNet* exactly match the sequence used by *glmnet*. These values can only be generated by descending logarithmically from λ_{zero} to $10^{-4} \cdot \lambda_{zero}$, in 100 steps. Hence, we can conclude that *glmnet* truncated its λ sequence from 100 to 66 values. Unfortunately, this behaviour is not explained in [4] or in the online documentation for *glmnet*.

6.2 Runtime: glmnet vs ElasticNet

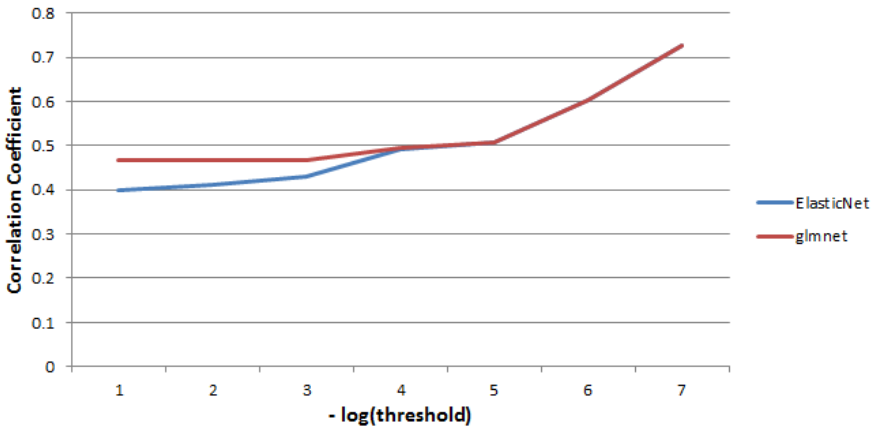
On average, a single experimental run (for a specific value of α) took 40 minutes. The MLR classifier's share of the runtime was 17 minutes, and *ElasticNet* needed 23 minutes. This means *glmnet* is, on average, approximately 35% faster than *ElasticNet*.

Interestingly, *ElasticNet* is actually faster than the MLR classifier for small datasets with less than 1000 instances. The most likely reason is that the R/WEKA integration introduces a small overhead, especially when performing (external) cross validation. Note that *glmnet* when executed in the R environment is always faster than *ElasticNet*.

6.3 Effect of modifying the convergence threshold

The predictive performance of *glmnet* (via the MLR classifier) significantly differs from the predictive performance of *ElasticNet* for higher thresholds as shown in Figure 6.1 below:

Figure 6.1: Correlation coefficient vs $-log(threshold)$



Note that the correlation coefficient for *glmnet* is exactly the same as that of *ElasticNet* for lower thresholds, but there are significant differences at higher thresholds. Also, note that the correlation coefficient for *glmnet* appears to be constant at higher thresholds. This goes against intuition because increasing the threshold should lower the predictive accuracy by making the algorithm converge prematurely.

To investigate this, we first observe that the MLR classifier runs an R function called *cv.glmnet*(), which is provided by the package *glmnet*. This function performs a cross validated fit over a sequence of λ values. The solution with the lowest cross validation error is chosen as the final model. Let λ_{best} be the λ corresponding to the lowest error.

The package also provides another function called *glmnet*(). It performs a fit over a sequence of λ values using Pathwise Coordinate Descent, and simply chooses the solution corresponding to the highest λ value in the sequence.

Thus, *glmnet*() should produce the same model as *cv.glmnet*() if the sequence provided to *glmnet*() is $(\lambda_{best}, 0)$. From my experiments, I have noted that this is not the case for higher values of the threshold. Effectively, for higher thresholds, the two functions provided by the package contradict each other. We can speculate that this is because *cv.glmnet*() runs a minimum number of iterations before terminating. This would explain why the correlation coefficient is constant for higher values of the threshold.

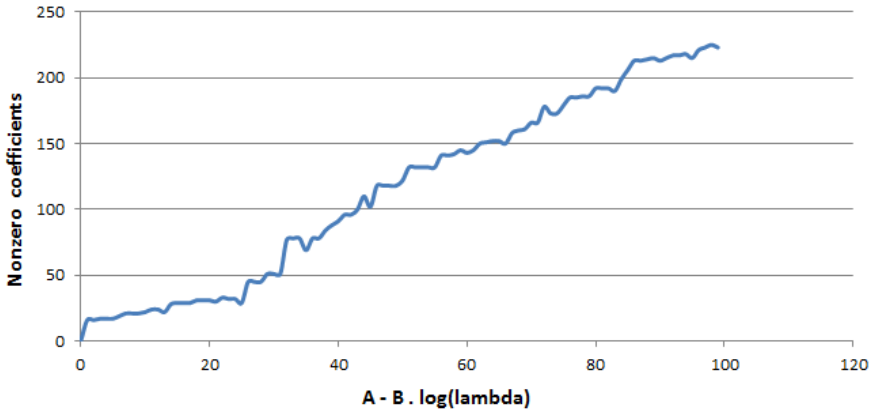
Note that this does not affect the credibility of the experiments performed earlier, because the value of the threshold was set to 1e-7 (by default), and this is low enough for *cv.glmnet*() to behave similarly to *glmnet*(). Finally, I have observed that the solutions produced by the function *glmnet*() are always close to the solutions obtained by *ElasticNet*, regardless of the threshold value.

6.4 Coefficient Shrinkage

To observe the coefficient shrinkage, I generated the dataset "cpu-edited.arff". This dataset was derived from "cpu.arff" by adding 220 extra variables. Each of these variables was generated from a linear combination of the original predictors plus a Gaussian noise component. Hence, most of the predictors are highly correlated with each other. This dataset also has more predictors than instances ($p > n$).

The *ElasticNet* algorithm was run on this dataset, with $\alpha = 0.001$ and $numModels = 100$. Recall that this builds a sequence of λ values logarithmically descending from λ_{zero} to λ_{min}, in 100 steps. It then builds a solution path with this λ sequence. The following graph shows the number of nonzero coefficients in each solution w.r.t the corresponding λ value:

Figure 6.2: Nonzero coefficients vs $(A - B \cdot log(\lambda))$



Here, A and B are constants that depend on the model parameters. Note that the graph validates one of the ideas explained in Section 4.3: As predicted, the number of nonzero coefficients in the model decreases as the we increase the value of λ.

However, we see that the function is not monotonic. A closer look at the coefficient vectors in the solution path reveals that there are some predictors whose coefficients are always either 0 or extremely close to 0. Obviously, these predictors must be loosely correlated with the residual. A specific λ value can sometimes affect these predictors a little more than the neighboring λ values. This explains why the function is slightly non-monotonic.

Interestingly, there appears to be a linear relationship between $log(\lambda)$ and the number of nonzero coefficients in the corresponding solution. It may be useful to investigate this further.

Chapter 7

Conclusion

In this project, a machine learning algorithm has been implemented in WEKA to solve the "elastic net" problem for linear regression. The WEKA algorithm is called *ElasticNet*, and it will soon be available as a WEKA package.

The elastic net problem involves finding an optimal coefficient vector to minimize an objective function roughly described as: (total squared error + penalty term). Here, the penalty is a combination of the L1 and L2 norms of the coefficients, and it can be tuned by a couple of parameters. With the right choice of parameters, solving the elastic net problem is equivalent to performing attribute selection.

The algorithm *ElasticNet* uses Cyclic Coordinate Descent to solve the elastic net problem. Cyclic Coordinate Descent can be a very fast procedure because the time taken for each coefficient update is linear in the number of predictors (or instances, if we choose Method 1 discussed in Section 5.1). The ideal coefficient update can be derived through calculus.

The WEKA implementation, called *ElasticNet*, is only slightly slower than *glmnet*, which is an R package that solves the elastic net problem. Note that WEKA uses Java whereas *glmnet* is written in FORTRAN. The results obtained by *ElasticNet* are mostly similar to *glmnet*, with a few notable exceptions. The experiments have uncovered a possible bug in the R/WEKA integration, and two possible bugs within the package *glmnet*.

Appendix A

Minimization by soft thresholding

Suppose we need to minimize the function $F(x) = ax^2 - bx + c|x| + d$, where $a > 0$ and $c \geq 0$.

It is clear that $F(x)$ never approaches $-\infty$ and so must have a global minimum x_{min}

Note that $F(x)$ is not differentiable at $x = 0$, but this could be the global minimum.

We now consider three cases, knowing that at least one of them is correct.

Case 1: Assume $x_{min} = 0$

We note that the value of $F(x = 0) = d$.

For Case 1 to hold true, d should be the lowest value of $F(x)$.

Case 2: Assume $x_{min} > 0$

$F(x) = ax^2 - bx + cx + d$

The derivative is $F^{'}(x) = 2ax - b + c$. At x_{min}, $F^{'}(x) = 0$. So,

$2ax_{min} - b + c = 0$

$x_{min} = \frac{b-c}{2a}$

Note that $x_{min} > 0$ only if $b > c$. Otherwise our assumption is violated.

Let $Min_2 = F(\frac{b-c}{2a})$ with $b > c$.

$Min_2 = a\frac{(b-c)^2}{4a^2} - (b-c)\frac{b-c}{2a} + d$

$Min_2 = d - \frac{(b-c)^2}{4a}$

Note that $b > c$ implies $Min_2 < d$. And this violates Case 1.

Case 3: Assume $x_{min} < 0$

$F(x) = ax^2 - bx - cx + d$

The derivative is $F^{'}(x) = 2ax - b - c$. At x_{min}, $F^{'}(x) = 0$. So,

$2ax_{min} - b - c = 0$

$x_{min} = \frac{b+c}{2a}$

Note that $x_{min} < 0$ only if $b < -c$. Otherwise our assumption is violated.

Let $Min_3 = F(\frac{b+c}{2a})$ with $b < -c$.

$Min_3 = a\frac{(b+c)^2}{4a^2} - (b+c)\frac{b+c}{2a} + d$

$Min_3 = d - \frac{(b+c)^2}{4a}$

Note that $b < -c$ implies $Min_3 < d$. And this violates Case 1.

Analysis

First, recall that at least one of the Cases 1-3 must hold true since $F(x)$ has a global minimum.

We note that the conditions for Cases 2 and 3, $b > c$ and $b < -c$, are mutually exclusive.

Hence, if $b > c$, both Cases 1 and 3 are violated, and so Case 2 must be correct.

Similarly, if $b < -c$, both Cases 1 and 2 are violated, and so Case 3 must be correct.

Lastly, note that if $-c \leq b \leq c$, both Cases 2 and 3 are violated, and so Case 1 must be correct.

So, only one of these Cases can be true, and the global minimum occurs at exactly one point.

To summarize, we can find the global minimum as follows:

$x_{min} = \frac{b-c}{2a}$ iff $b > c$

$x_{min} = \frac{b+c}{2a}$ iff $b < -c$

$x_{min} = 0$ iff $-c \leq b \leq c$

We now define the Soft Thresholding operator $S(p, q)$ with $q \geq 0$ as follows:

$S(p, q) = p - q$ iff $p > q$

$S(p, q) = p + q$ iff $p < -q$

$S(p, q) = 0$ iff $-q \leq p \leq q$

Hence, we can say that $x_{min} = \dfrac{S(b, c)}{2a}$

Appendix B

Generalized coefficient update

We have $F_{objective} = \frac{1}{2}\frac{\sum_{i=1}^{n} w_i(y_i - x_i^T\beta)^2}{\sum_{i=1}^{n} w_i} + \lambda\sum_{j=1}^{p}(\frac{1}{2}(1-\alpha)\beta_j^2 + \alpha|\beta_j|)$

In order to minimize $F_{objective}$ w.r.t a specific β_j, we must rewrite it in terms of β_j.

First we define r_i^{-j} as the i^{th} residual obtained by setting $\beta_j = 0$.

Clearly, the i^{th} residual $r_i = y_i - \sum_{k=1}^{p} x_{ik}\beta_k = r_i^{-j} - x_{ij}\beta_j$. So,

$F_{objective} = \frac{1}{2\sum_{i=1}^{n} w_i}\sum_{i=1}^{n} w_i(r_i^{-j} - x_{ij}\beta_j)^2 + \frac{1}{2}\lambda(1-\alpha)\beta_j^2 + \lambda\alpha|\beta_j| + C$

Here, C is a collection of terms independent of β_j. The equation can be rewritten as follows:

$F_{objective} = \frac{1}{2\sum_{i=1}^{n} w_i}\sum_{i=1}^{n} w_i(r_i^{-j^2} + x_{ij}^2\beta_j^2 - 2r_i^{-j}x_{ij}\beta_j) + \frac{1}{2}\lambda(1-\alpha)\beta_j^2 + \lambda\alpha|\beta_j| + C$

Clearly, $\sum_{i=1}^{n} w_i r_i^{-j^2}$ is independent of β_j. So,

$F_{objective} = \frac{1}{2\sum_{i=1}^{n} w_i}(\beta_j^2\sum_{i=1}^{n} w_i x_{ij}^2 - 2\beta_j\sum_{i=1}^{n} w_i r_i^{-j}x_{ij}) + \frac{1}{2}\lambda(1-\alpha)\beta_j^2 + \lambda\alpha|\beta_j| + C$

$F_{objective} = \frac{1}{2}(\frac{\sum_{i=1}^{n} w_i x_{ij}^2}{\sum_{i=1}^{n} w_i} + \lambda(1-\alpha))\cdot\beta_j^2 - \frac{\sum_{i=1}^{n} w_i r_i^{-j}x_{ij}}{\sum_{i=1}^{n} w_i}\cdot\beta_j + \lambda\alpha|\beta_j| + C$

We note that $F_{objective}$ is of the form $(a\beta_j^2 - b\beta_j + c|\beta_j| + d)$, where $a \geq 0$ and $c \geq 0$.

If we assume $a > 0$, then minimizing β_j is similar to the problem solved in Appendix A. So, $\beta_j^{optimal} = \frac{S(\frac{\sum_{i=1}^{n} w_i r_i^{-j}x_{ij}}{\sum_{i=1}^{n} w_i}, \lambda\alpha)}{2\cdot\frac{1}{2}(\frac{\sum_{i=1}^{n} w_i x_{ij}^2}{\sum_{i=1}^{n} w_i} + \lambda(1-\alpha))}$. This reduces to:

$\beta_j^{optimal} = \frac{S(\sum_{i=1}^{n} w_i r_i^{-j}x_{ij}, \lambda\alpha\sum_{i=1}^{n} w_i)}{\sum_{i=1}^{n} w_i x_{ij}^2 + \lambda(1-\alpha)\sum_{i=1}^{n} w_i}$. This is the coefficient update formula used in this project.

We now note that $a = 0$ means $\sum_{i=1}^{n} w_i x_{ij}^2 = 0$, since $\lambda(1-\alpha) \geq 0$.

Obviously, $\sum_{i=1}^{n} w_i x_{ij}^2 = 0$ means $x_{ij} = 0$ for $i \in \{1, 2, ...n\}$.

This would mean $\sum_{i=1}^{n} w_i r_i^{-j}x_{ij} = 0$, hence $b = \frac{\sum_{i=1}^{n} w_i r_i^{-j}x_{ij}}{\sum_{i=1}^{n} w_i} = 0$. So,

$F_{objective} = \lambda\alpha|\beta_j| + C$

Clearly, the global minimum of this function is at $\beta_j = 0$.

Hence, if $a = \frac{1}{2}(\frac{\sum_{i=1}^{n} w_i x_{ij}^2}{\sum_{i=1}^{n} w_i} + \lambda(1-\alpha)) = 0$ for some j, then for every iteration, $\beta_j^{optimal} = 0$

Appendix C

Naive and covariance updates

C.1 Method 1: Naive updates

Let r_i be the i^{th} residual and r_i^{-j} be the i^{th} residual obtained by setting $\beta_j = 0$

To update β_j, we first need to compute $F_j = \sum_{i=1}^{n} w_i r_i^{-j} x_{ij}$

Clearly, $r^{-j} = r_i + x_{ij}\beta_j$. So,

$F_j = \sum_{i=1}^{n} w_i x_{ij}(r_i + x_{ij}\beta_j)$

$F_j = \sum_{i=1}^{n} w_i x_{ij} r_i + \beta_j \sum_{i=1}^{n} w_i x_{ij}^2$

C.2 Method 2: Covariance updates

Let r_i be the i^{th} residual and r_i^{-j} be the i^{th} residual obtained by setting $\beta_j = 0$

We need to find $F_j = \sum_{i=1}^{n} w_i r_i^{-j} x_{ij}$. From C.1, we see that:

$F_j = \sum_{i=1}^{n} w_i x_{ij} r_i + \beta_j \sum_{i=1}^{n} w_i x_{ij}^2$

Let $G_j = \sum_{i=1}^{n} w_i x_{ij} r_i$

We have $r_i = y_i - \sum_{k=1}^{p} x_{ik}\beta_k$. So,

$G_j = \sum_{i=1}^{n} w_i x_{ij}(y_i - \sum_{k=1}^{p} x_{ik}\beta_k)$

$G_j = \sum_{i=1}^{n} w_i x_{ij} y_i - \sum_{i=1}^{n}(w_i x_{ij} \sum_{k=1}^{p} x_{ik}\beta_k)$

The second term can be rewritten as follows:

$\sum_{i=1}^{n}(w_i x_{ij} \sum_{k=1}^{p} x_{ik}\beta_k) = \sum_{k=1}^{p}(\beta_k \sum_{i=1}^{n} w_i x_{ij} x_{ik})$. Thus,

$G_j = \sum_{i=1}^{n} w_i x_{ij} y_i - \sum_{k=1}^{p}(\beta_k \sum_{i=1}^{n} w_i x_{ij} x_{ik})$

Finally, $F_j = \sum_{i=1}^{n} w_i x_{ij} y_i - \sum_{k=1}^{p}(\beta_k \sum_{i=1}^{n} w_i x_{ij} x_{ik}) + \beta_j \sum_{i=1}^{n} w_i x_{ij}^2$

Appendix D

Squared error update

We have $F_{error} = \frac{1}{2}\frac{\sum_{i=1}^{n} w_i(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2}{\sum_{i=1}^{n} w_i}$

$F_{error} = \frac{1}{2\sum_{i=1}^{n} w_i}\sum_{i=1}^{n} w_i(y_i^2 + (\sum_{j=1}^{p} x_{ij}\beta_j)^2 - 2y_i\sum_{j=1}^{p} x_{ij}\beta_j)$

To update F_{error} without computing residuals, we must write it in terms of β_j.

First, we should expand $(\sum_{j=1}^{p} x_{ij}\beta_j)^2$ and extract the terms with β_j.

For example, $(x+m+n)^2 = x^2+m^2+n^2+2xm+2xn+2mn = x^2+2x(m+n)+m^2+n^2+2mn$

We can use a similar logic to isolate the terms with β_j from the expansion of $(\sum_{j=1}^{p} x_{ij}\beta_j)^2$

For this, we define the set $P^{-j} = \{1, 2, ...p\} - \{j\}$ where $1 \leq j \leq p$

In terms of β_j, we have $(\sum_{j=1}^{p} x_{ij}\beta_j)^2 = x_{ij}^2\beta_j^2 + 2x_{ij}\beta_j\sum_{k\in P^{-j}} x_{ik}\beta_k + Other\ terms$

Substituting this in F_{error}, and separating the terms independent of β_j, we get:

$F_{error} = \frac{1}{2\sum_{i=1}^{n} w_i}\sum_{i=1}^{n} w_i(x_{ij}^2\beta_j^2 + 2x_{ij}\beta_j\sum_{k\in P^{-j}} x_{ik}\beta_k) - \frac{1}{\sum_{i=1}^{n} w_i}\sum_{i=1}^{n}(w_i y_i x_{ij}\beta_j) + C$

Here, C is a collection of terms independent of β_j. The equation can be rewritten as follows:

$F_{error} = \beta_j^2 \cdot \frac{\sum_{i=1}^{n} w_i x_{ij}^2}{2\sum_{i=1}^{n} w_i} + \beta_j \cdot \frac{1}{\sum_{i=1}^{n} w_i}\sum_{i=1}^{n}(w_i x_{ij}\sum_{k\in P^{-j}} x_{ik}\beta_k) - \beta_j \cdot \frac{\sum_{i=1}^{n} w_i y_i x_{ij}}{\sum_{i=1}^{n} w_i} + C$

We know that $\sum_{i=1}^{n}(w_i x_{ij}\sum_{k\in P^{-j}} x_{ik}\beta_k) = \sum_{k\in P^{-j}}(\beta_k\sum_{i=1}^{n} w_i x_{ij}x_{ik})$. So,

$F_{error} = \beta_j^2 \cdot \frac{\sum_{i=1}^{n} w_i x_{ij}^2}{2\sum_{i=1}^{n} w_i} + \beta_j \cdot \frac{1}{\sum_{i=1}^{n} w_i}\sum_{k\in P^{-j}}(\beta_k\sum_{i=1}^{n} w_i x_{ij}x_{ik}) - \beta_j \cdot \frac{\sum_{i=1}^{n} w_i y_i x_{ij}}{\sum_{i=1}^{n} w_i} + C$

$F_{error} = \beta_j^2 \cdot \frac{\sum_{i=1}^{n} w_i x_{ij}^2}{2\sum_{i=1}^{n} w_i} - \beta_j \cdot \frac{1}{\sum_{i=1}^{n} w_i}(\sum_{i=1}^{n} w_i y_i x_{ij} - \sum_{k\in P^{-j}}(\beta_k\sum_{i=1}^{n} w_i x_{ij}x_{ik})) + C$

From appendix C, we have $G_j = \sum_{i=1}^{n} w_i x_{ij} r_i = \sum_{i=1}^{n} w_i x_{ij} y_i - \sum_{k=1}^{p}(\beta_k\sum_{i=1}^{n} w_i x_{ij}x_{ik})$

If G_j has been updated with the new β_j, we can write:

$\sum_{i=1}^{n} w_i y_i x_{ij} - \sum_{k\in P^{-j}}(\beta_k\sum_{i=1}^{n} w_i x_{ij}x_{ik}) = G_j^{new} + \beta_j^{new}\sum_{i=1}^{n} w_i x_{ij}^2$. So,

$F_{error} = \beta_j^2 \cdot \frac{\sum_{i=1}^{n} w_i x_{ij}^2}{2\sum_{i=1}^{n} w_i} - \beta_j \cdot \frac{1}{\sum_{i=1}^{n} w_i}(G_j^{new} + \beta_j^{new}\sum_{i=1}^{n} w_i x_{ij}^2) + C$

Hence, the term to be added to update the squared error is:

$F_{error}^{new} - F_{error}^{old} = (\beta_j^{new^2} - \beta_j^{old^2}) \cdot \frac{\sum_{i=1}^{n} w_i x_{ij}^2}{2\sum_{i=1}^{n} w_i} - (\beta_j{}^{new} - \beta_j{}^{old}) \cdot \frac{1}{\sum_{i=1}^{n} w_i}(G_j^{new} + \beta_j^{new}\sum_{i=1}^{n} w_i x_{ij}^2)$

Appendix E

Accounting for uncentered variables

Here, we deal with cases where the predictors are not centered or the class is not standardized.

In such cases, it is possible to incorporate the scaling and centering into our computations.

We first define the weighted mean of the j^{th} predictor, $\mu_j = \frac{\sum_{i=1}^{n} w_i x_{ij}}{\sum_{i=1}^{n} w_i}$

Next, we define the weighted mean of the class, $\mu_{class} = \frac{\sum_{i=1}^{n} w_i y_i}{\sum_{i=1}^{n} w_i}$

Finally, we define the variance of the class, $\sigma^2_{class} = \frac{\sum_{i=1}^{n} w_i (y_i - \mu_{class})^2}{\sum_{i=1}^{n} w_i}$

To center the j^{th} predictor, we must replace x_{ij} with $(x_{ij} - \mu_j)$ for $i \in \{1, 2, ...n\}$

To standardize the class, we must replace y_i with $\frac{y_i - \mu_{class}}{\sigma_{class}}$ for $i \in \{1, 2, ...n\}$

E.1 Weighted sum of squares

Let $S_j = \sum_{i=1}^{n} w_i x_{ij}^{centered^2}$

We know that $x_{ij}^{centered} = x_{ij} - \mu_j$. So,

$S_j = \sum_{i=1}^{n} w_i (x_{ij} - \mu_j)^2$

$S_j = \sum_{i=1}^{n} w_i (x_{ij}^2 + \mu_j^2 - 2\mu_j x_{ij})$

$S_j = \sum_{i=1}^{n} w_i x_{ij}^2 + \mu_j^2 \sum_{i=1}^{n} w_i - 2\mu_j \sum_{i=1}^{n} w_i x_{ij}$

We know that $\sum_{i=1}^{n} w_i x_{ij} = \mu_j \sum_{i=1}^{n} w_i$. So,

$S_j = \sum_{i=1}^{n} w_i x_{ij}^2 + \mu_j^2 \sum_{i=1}^{n} w_i - 2\mu_j^2 \sum_{i=1}^{n} w_i$

$S_j = \sum_{i=1}^{n} w_i x_{ij}^2 - \mu_j^2 \sum_{i=1}^{n} w_i$

E.2 Dot product between class and predictor

Let $D_j = \sum_{i=1}^{n} w_i x_{ij}^{centered} y_i^{standardized}$

$D_j = \sum_{i=1}^{n} (w_i(x_{ij} - \mu_j) \cdot \frac{y_i - \mu_{class}}{\sigma_{class}})$

$D_j = \frac{1}{\sigma_{class}} \sum_{i=1}^{n} (w_i(x_{ij} - \mu_j)(y_i - \mu_{class}))$

$D_j = \frac{1}{\sigma_{class}} (\sum_{i=1}^{n} w_i x_{ij} y_i - \mu_j \sum_{i=1}^{n} w_i y_i - \mu_{class} \sum_{i=1}^{n} w_i x_{ij} + \mu_j \mu_{class} \sum_{i=1}^{n} w_i)$

We know that $\sum_{i=1}^{n} w_i y_i = \mu_{class} \sum_{i=1}^{n} w_i$, and $\sum_{i=1}^{n} w_i x_{ij} = \mu_j \sum_{i=1}^{n} w_i$. So,

$D_j = \frac{1}{\sigma_{class}} (\sum_{i=1}^{n} w_i y_i x_{ij} - \mu_j \mu_{class} \sum_{i=1}^{n} w_i - \mu_{class} \mu_j \sum_{i=1}^{n} w_i + \mu_j \mu_{class} \sum_{i=1}^{n} w_i)$

Thus, $D_j = \frac{1}{\sigma_{class}} (\sum_{i=1}^{n} w_i y_i x_{ij} - \mu_j \mu_{class} \sum_{i=1}^{n} w_i)$

E.3 Dot product between two predictors

Let $D_{jk} = \sum_{i=1}^{n} w_i x_{ij}^{centered} x_{ik}^{centered}$

$D_{jk} = \sum_{i=1}^{n} (w_i(x_{ij} - \mu_j)(x_{ik} - \mu_k))$

$D_{jk} = \sum_{i=1}^{n} w_i x_{ij} x_{ik} + \mu_j \mu_k \sum_{i=1}^{n} w_i - \mu_j \sum_{i=1}^{n} w_i x_{ik} - \mu_k \sum_{i=1}^{n} w_i x_{ij}$

We know that $\sum_{i=1}^{n} w_i x_{im} = \mu_m \sum_{i=1}^{n} w_i$ for any $m \in \{1, 2, ...p\}$. So,

$D_{jk} = \sum_{i=1}^{n} w_i x_{ij} x_{ik} + \mu_j \mu_k \sum_{i=1}^{n} w_i - \mu_j \mu_k \sum_{i=1}^{n} w_i - \mu_k \mu_j \sum_{i=1}^{n} w_i$

$D_{jk} = \sum_{i=1}^{n} w_i x_{ij} x_{ik} - \mu_j \mu_k \sum_{i=1}^{n} w_i$

E.4 Computing the class variance

We have $\sigma_{class}^2 = \frac{\sum_{i=1}^{n} w_i (y_i - \mu_{class})^2}{\sum_{i=1}^{n} w_i}$

$\sigma_{class}^2 = \frac{1}{\sum_{i=1}^{n} w_i} \sum_{i=1}^{n} w_i (y_i^2 + \mu_{class}^2 - 2 y_i \mu_{class})$

$\sigma_{class}^2 = \frac{1}{\sum_{i=1}^{n} w_i} (\sum_{i=1}^{n} w_i y_i^2 + \mu_{class}^2 \sum_{i=1}^{n} w_i - 2\mu_{class} \sum_{i=1}^{n} w_i y_i)$

We know that $\sum_{i=1}^{n} w_i y_i = \mu_{class} \sum_{i=1}^{n} w_i$. So,

$\sigma_{class}^2 = \frac{1}{\sum_{i=1}^{n} w_i} (\sum_{i=1}^{n} w_i y_i^2 + \mu_{class}^2 \sum_{i=1}^{n} w_i - 2\mu_{class}^2 \sum_{i=1}^{n} w_i)$

$\sigma_{class}^2 = \frac{1}{\sum_{i=1}^{n} w_i} (\sum_{i=1}^{n} w_i y_i^2 - \mu_{class}^2 \sum_{i=1}^{n} w_i)$

Hence, $\sigma_{class}^2 = \frac{\sum_{i=1}^{n} w_i y_i^2}{\sum_{i=1}^{n} w_i} - \mu_{class}^2$

Appendix F

Formulas for uncentered residuals

Let r_i be the i^{th} residual.

We have $r_i = y_i - \sum_{j=1}^{p} \beta_j x_{ij}^{centered}$

We define the uncentered residual $r_i^{'} = y_i - \sum_{j=1}^{p} \beta_j x_{ij}$

We know that $x_{ij}^{centered} = x_{ij} - \mu_j$. So,

$r_i = y_i - \sum_{j=1}^{p} \beta_j (x_{ij} - \mu_j)$

$r_i = y_i - \sum_{j=1}^{p} \beta_j x_{ij} + \sum_{j=1}^{p} \beta_j \mu_j$

Hence, $r_i = r_i^{'} + \sum_{j=1}^{p} \beta_j \mu_j$

F.1 Dot product between residual and predictor

We need to calculate $G_j = \sum_{i=1}^{n} w_i r_i x_{ij}^{centered}$

We know that $x_{ij}^{centered} = x_{ij} - \mu_j$. So,

$G_j = \sum_{i=1}^{n} w_i r_i (x_{ij} - \mu_j)$

$G_j = \sum_{i=1}^{n} w_i r_i x_{ij} - \mu_j \sum_{i=1}^{n} w_i r_i$

We also know that $r_i = r_i^{'} + \sum_{j=1}^{p} \beta_j \mu_j$. So,

$G_j = \sum_{i=1}^{n} (w_i x_{ij} (r_i^{'} + \sum_{j=1}^{p} \beta_j \mu_j)) - \mu_j \sum_{i=1}^{n} w_i (r_i^{'} + \sum_{j=1}^{p} \beta_j \mu_j)$

We know that $\sum_{j=1}^{p} \beta_j \mu_j$ is independent of i. So,

$G_j = \sum_{i=1}^{n} w_i x_{ij} r_i^{'} + \sum_{j=1}^{p} \beta_j \mu_j \cdot \sum_{i=1}^{n} w_i x_{ij} - \mu_j \sum_{i=1}^{n} w_i r_i^{'} - \sum_{j=1}^{p} \beta_j \mu_j \cdot \mu_j \sum_{i=1}^{n} w_i$

The second and fourth terms cancel out since $\sum_{i=1}^{n} w_i x_{ij} = \mu_j \sum_{i=1}^{n} w_i$. Hence,

$G_j = \sum_{i=1}^{n} w_i x_{ij} r_i^{'} - \mu_j \sum_{i=1}^{n} w_i r_i^{'}$

F.2 Squared error in terms of uncentered residuals

We have $F_{error} = \frac{1}{2}\frac{\sum_{i=1}^{n} w_i(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2}{\sum_{i=1}^{n} w_i}$. This can be rewritten as:

$F_{error} = \frac{1}{2\sum_{i=1}^{n} w_i}\sum_{i=1}^{n} w_i r_i^2$, where r_i is the i^{th} residual.

Now, we know that $r_i = r_i^{'} + \sum_{j=1}^{p} \beta_j \mu_j$, where $r_i^{'}$ is the i^{th} uncentered residual. So,

$F_{error} = \frac{1}{2\sum_{i=1}^{n} w_i}\sum_{i=1}^{n} w_i(r_i^{'} + \sum_{j=1}^{p} \beta_j \mu_j)^2$

$F_{error} = \frac{1}{2\sum_{i=1}^{n} w_i}\sum_{i=1}^{n} w_i(r_i^{'2} + 2r_i^{'}\sum_{j=1}^{p} \beta_j \mu_j + (\sum_{j=1}^{p} \beta_j \mu_j)^2)$

We know that $\sum_{j=1}^{p} \beta_j \mu_j$ is independent of i. So,

$F_{error} = \frac{1}{2\sum_{i=1}^{n} w_i}(\sum_{i=1}^{n} w_i r_i^{'2} + 2\sum_{j=1}^{p} \beta_j \mu_j \cdot \sum_{i=1}^{n} w_i r_i^{'} + (\sum_{j=1}^{p} \beta_j \mu_j)^2 \cdot \sum_{i=1}^{n} w_i)$

$F_{error} = \frac{1}{2\sum_{i=1}^{n} w_i}(\sum_{i=1}^{n} w_i r_i^{'2} + 2\sum_{j=1}^{p} \beta_j \mu_j \cdot \sum_{i=1}^{n} w_i r_i^{'}) + \frac{(\sum_{j=1}^{p} \beta_j \mu_j)^2}{2}$

Appendix G

R code for running glmnet

```
# import libraries
library(foreign)
library(glmnet)

# Function to run cv.glmnet() and
# 1. Compute the correlation coefficient and
# 2. Return the default lambda sequence
funct2 = function(path,al,thr=1e-7) {

# Modify path to directory with arff files
path = paste("C:/Users/Nikhil/Desktop/Data_Regression/",path,".arff",sep="")

# Read the data
data = read.arff(path)

# Number of rows and columns
rows=length(data[[1]])
totalcols=length(data)
cols=totalcols-1

# Put all training data in a list column wise
lst = c()
for(i in 1:cols) { lst = c(lst,data[[i]]) }

# Reshape data into training matrix
labels = data[[totalcols]]
trainmat = array(lst, dim=c(rows,cols))

# Fit glmnet object with given params
fit = cv.glmnet(x=trainmat,y=labels,standardize=FALSE,alpha=al,thresh=thr)

# Test the best model on the training data
minlam = fit$lambda.min
pred = predict(fit,trainmat,s=minlam)
```

```
corrcoef=0; sumActual=0; sumPred=0;
sumsqActual=0; sumsqPred=0; dotprod=0

# Calculate correlation coefficient
for(i in 1:rows) {
        sumActual = sumActual + labels[[i]]
        sumsqActual = sumsqActual + labels[[i]]*labels[[i]]
        sumPred = sumPred + pred[[i]]
        sumsqPred = sumsqPred + pred[[i]]*pred[[i]]
        dotprod = dotprod + labels[[i]]*pred[[i]]
}

corrcoef = (rows*dotprod-sumActual*sumPred)/
        sqrt((rows*sumsqActual-sumActual*sumActual)*(rows*sumsqPred-sumPred*sumPred))

# Generated lambda sequence
lams = fit$lambda

# Create output list
retobj = list(corr=corrcoef,lams=lams)
return(retobj)
}

# Usage
# Runs the algorithm over "Dataset.arff" with alpha = 1 and threshold = 1e-7
obj = funct2(path="Dataset",al=1)

# Print correlation coefficient
obj$corr

# Print lambda sequence
obj$lams
```

Bibliography

[1] Charlotte Møller Andersen and Rasmus Bro. Variable selection in regression—a tutorial. *Journal of Chemometrics*, 24(11-12):728–737, 2010.

[2] Leo Breiman. Better subset regression using the nonnegative garrote. *Technometrics*, 37(4):373–384, 1995.

[3] Bradley Efron, Trevor Hastie, Iain Johnstone, Robert Tibshirani, et al. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

[4] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 33(1):1, 2010.

[5] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[6] Trevor Hastie, Jonathan Taylor, Robert Tibshirani, Guenther Walther, et al. Forward stagewise regression and the monotone lasso. *Electronic Journal of Statistics*, 1:1–29, 2007.

[7] Claude Nadeau and Yoshua Bengio. Inference for the generalization error. *Machine Learning*, 52(3):239–281, 2003.

[8] Ming Yuan and Yi Lin. On the non-negative garrotte estimator. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 69(2):143–161, 2007.